

K: AN OBJECT-ORIENTED KNOWLEDGE-BASE PROGRAMMING LANGUAGE
FOR SOFTWARE DEVELOPMENT AND PROTOTYPING

By

YUH-MING SHYY

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1992



ACKNOWLEDGEMENTS

I am grateful to a number of individuals for their contributions during the development of this dissertation. In particular, I would like to thank my advisor, Prof. Stanley Y.W. Su, for his continuous encouragement, guidance, and support throughout the course of this research work. I would also like to thank Prof. Herman Lam, Prof. Sharma Chakravarthy, Prof. Manuel Bermudez, and Prof. Thomas Bullock for their participation on my supervisory committee and careful review of my dissertation.

My special appreciation goes to my colleague Javier Arroyo for his many fruitful suggestions and the implementation of the K.1 prototype. I also thank Sharon Grant, the data-base center secretary, for her constant help.

Finally, and most importantly, I would like to thank my dear wife, Ching-Jung Wu, and my parents, Yo-Ching Shyy and Ching-Mei Lee Shyy, for their encouragement, patience, and understanding during the period of my Ph.D. study.

This research was supported by National Science Foundation Grant #DMC-8814989 and Florida High Technology and Industrial Council Grant #UPN90090708.

TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGEMENTS	ii
ABSTRACT	v
CHAPTERS	
1 INTRODUCTION	1
1.1 Motivation	1
1.2 Language Design Principles	4
1.3 System Overview	6
1.4 Dissertation Organization	9
2 SURVEY OF RELATED WORKS	13
2.1 Data-base Programming Language	13
2.2 Structural and Behavioral Modeling	16
2.3 KBMS-supported Software Development System	18
3 LANGUAGE OVERVIEW	21
3.1 Knowledge Abstractions	21
3.2 Model Extensibility and Reflexivity	27
3.3 Persistence	30
3.4 Type System	34
3.5 KBMS Operations	37
4 STRUCTURAL ABSTRACTION MECHANISMS	45
4.1 Association Definition	46
4.2 Association Pattern	51
4.3 Context Looping Statement and Navigation	62
4.4 Existential and Universal Quantifiers	69
5 BEHAVIORAL ABSTRACTION MECHANISMS	75
5.1 Method Definition	75
5.2 Rule Definition	84

6	COMPUTATION MODEL	92
6.1	Overview	92
6.2	Extended Object-oriented Computation . . .	95
7	KBMS-SUPPORTED EVOLUTIONARY PROTOTYPING	102
7.1	Overview	102
7.2	Method Model and Control Associations . . .	107
8	SYSTEM ARCHITECTURE AND IMPLEMENTATION	123
8.1	System Architecture	123
8.2	Implementation: Mapping from K.1 to C++ . .	128
9	CONCLUSION AND FUTURE RESEARCH DIRECTIONS . . .	142
9.1	Conclusion	142
9.2	Future Research Directions	145
APPENDICES		
A	SYNTAX SUMMARY OF K	152
B	PARTS KNOWLEDGE-BASE EXAMPLE	160
REFERENCES		166
BIOGRAPHICAL SKETCH		177

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

K: AN OBJECT-ORIENTED KNOWLEDGE-BASE PROGRAMMING LANGUAGE
FOR SOFTWARE DEVELOPMENT AND PROTOTYPING

By

Yuh-Ming Shyy

August, 1992

Chairperson: Dr. Stanley Y.W. Su

Major Department: Computer and Information Sciences

The OSAM*.KBMS is a research prototype of a knowledge-base management system, or the so-called "next generation data-base management system," for nontraditional data- and knowledge-intensive applications. In order to solve the "impedance mismatch" problems between data-base languages (which include data definition languages, query languages, and rule languages) and traditional programming languages, we have developed an object-oriented knowledge-base programming language called K to serve as the high-level interface of OSAM*.KBMS for defining, querying, and manipulating the knowledge base, as well as to write codes to implement any application system. In addition to the well-known object-oriented virtues such as abstract data types, information hiding, complex objects, relationships, inheritance, reusable codes, etc., K provides six important features.

(1) Powerful abstraction mechanisms for supporting the underlying knowledge model which captures any application domain knowledge in terms of the structural associations (such as generalization and aggregation), functional associations (such as the client-server relationship), methods, and knowledge rules (such as constraints and triggers).

(2) A strong notation of address-independent object identifiers (oid) instead of physical pointers.

(3) A persistence mechanism for supporting both persistent and transient objects uniformly.

(4) A flexible type system that supports both static type checking and multiple views of objects in multiple classes.

(5) A declarative knowledge retrieval mechanism based on object association patterns for querying the knowledge base.

(6) Multi-paradigm programming constructs for specifying procedural and rule-based computations.

K can be used for both the specification and implementation of any application system to any level of detail and therefore also facilitates KBMS-supported evolutionary prototyping of software systems. This dissertation presents the design and implementation of K in terms of its underlying knowledge model, linguistic facilities, implementation architecture, and application to evolutionary prototyping.

CHAPTER 1 INTRODUCTION

1.1 Motivation

With a view to widening the applicability of data-base technology to nontraditional application domains such as Computer-Aided Software Engineering (CASE), Computer-Aided Design and Manufacturing (CAD/CAM), Office Information Systems, and Knowledge Representation Systems, many so-called "Next-Generation Data-Base Management Systems (DBMS)" [ATK90, SIG90, ACM91] have been proposed in recent years. In general, a next-generation DBMS extends the functionalities of traditional DBMSs (such as persistent data management, query processing, concurrency control, and recovery) in either or both of the following aspects. Firstly, object-oriented data modeling constructs are introduced to model complex application domains and the behavioral specifications are also incorporated into the domain and functionality of a DBMS in terms of user-defined methods. Secondly, rule management facilities are introduced to manage and process a large number of knowledge rules. Usually, the structure and behavior of a complex system are often subject to design, operation, and system rules. If these rules can be explicitly specified in

an application system, they can be used for automatically maintaining system constraints and/or triggering predefined actions when certain events occur. Note that, although the semantics represented by rules can be implemented in methods, high-level declarative rules make it much easier for a data-base designer to clearly capture the semantics and thus simplify the tasks of implementation, debugging, and maintenance. In order to define, query, and manipulate the data-base and to avoid the impedance mismatch problems [COP84, MAI89] between data-base languages (which include data definition languages, query languages, and/or rule languages) and traditional programming languages, next-generation data-base programming languages are also needed. In this dissertation, we shall use the term "knowledge-base management system" (KBMS) and "knowledge-base programming language" (KBPL) to refer such a next-generation data-base management system and data-base programming language, respectively.

The OSAM*.KBMS [LAM89a,b,YAS91] is a research prototype of KBMS that is based on an object-oriented semantic association model OSAM* [SU86,89a,b,92] and developed at the Database Systems Research and Development Center of the University of Florida. In the past several years, an object-oriented query language (OQL) [ALA89, GU091] and constraint language [ALA90, SU91] have been developed for specifying queries and rules. However, the implementation of methods still needs to be done in such traditional programming

languages as C++ [STR86]. Because the implementation language does not directly support the OSAM* knowledge model, all of the classical impedance mismatch problems still exist. To solve the problems, we have developed a single integrated object-oriented knowledge-base programming language called K [SHY91] to serve as the high-level interface of OSAM*.KBMS for defining, querying, and manipulating the knowledge-base, as well as to code methods of any data/knowledge-intensive application system. In addition to such well-known object-oriented virtues as abstract data types, information hiding, complex objects, relationships, inheritance, reusable codes, etc., K provides the following six important features.

(1) Powerful abstraction mechanisms for supporting the underlying knowledge model which captures any application domain knowledge in terms of the structural associations (such as generalization and aggregation), functional associations (such as the client-server relationship), methods, and knowledge rules (such as constraints and triggers) in an integrated fashion.

(2) A strong notation of address-independent object identifiers (oid) instead of physical pointers.

(3) A persistence mechanism for supporting both persistent and transient objects uniformly.

(4) A flexible type system which supports both static type checking and multiple representations of objects in multiple classes.

(5) A declarative knowledge retrieval mechanism based on object association patterns for querying the knowledge base.

(6) Basic data structures (set, list, and array) and multi-paradigm programming constructs for specifying procedural and rule-based computations.

K can be used for both the specification and implementation of an application system to any level of detail and therefore facilitates KBMS-supported evolutionary prototyping [SU92], which will be described in Chapter 7.

1.2 Language Design Principles

The design of K is guided by the following general design principles. More specific design rationales will be given throughout this dissertation.

(1) Direct support of the OSAM*.KBMS kernel knowledge model. K should provide the knowledge abstraction mechanisms to support the extensible and reflexive OSAM*.KBMS kernel knowledge model, which will be described in Chapter 3. All the semantic constructs such as classes, associations, methods, and rules should be treated as first class objects in the same way as any other objects in K.

(2) Wide-spectrum for both specification and implementation. K should be a uniform language for knowledge definition, knowledge retrieval, knowledge manipulation, and general computation with persistent/transient objects.

(3) Computationally complete. K should provide all of the basic data structures (set, array, and list), control structures (sequence, repetition, and condition), and rule specification constructs for the users to implement any algorithm and to perform any computation.

(4) Maintainability and readability. The software written in K should have readable syntax and stable semantics so that it can be easily understood and maintained.

(5) Seamless incorporation of query/rule language. Instead of just embedding the existing query and rule language of OSAM*.KBMS into K, a uniform and well-integrated syntax is necessary to provide set-oriented and declarative query and rule specification facilities without any conflict or ambiguity with other programming constructs of K. New constructs should be introduced only if we can demonstrate one or more of the following points: readability, new concept, and conciseness. Besides, new constructs must satisfy the orthogonality principle, i.e., any combination of the programming constructs is allowed.

(6) Strongly typed. As K is to be used for the development of complex software systems, it should be a strongly typed language so that as many type errors as possible can be checked by static type checking at compile time. On the other hand, the type system should be flexible enough to support the distributed view, or multiple

representations, of OSAM* objects, as will be discussed in Chapter 3.

(7) More emphasis on functionalities rather than efficiency. As a high-level programming language, K should put more emphasis on its functionalities than efficiency so that complex application systems can be rapidly constructed by the use of those high-level facilities of K. With the rate of hardware progress, we do not feel that efficiency will be a serious concern in the future.

1.3 System Overview

K is part of a KBMS-supported software development system whose layer structure is shown in Figure 1.1. Starting from the middle layer, i.e., the abstraction layer, we have developed an extensible and reflexive object-oriented knowledge model which (i) provides powerful abstraction mechanisms for explicitly capturing the structural and behavioral properties of all software systems and the application domain objects that these systems deal with in terms of the structural associations, methods, and knowledge rules of object classes, (ii) reflexively models itself as a kernel model so that all the meta information (i.e., structural and behavioral properties of objects) can also be modeled as object classes. This knowledge model serves as the underlying model of K, which is represented by the language layer. In order to establish the proper computing environment

for the developers, a set of well-integrated tools will be provided at the environment layer for the users to (i) define, browse, modify, query, and test the software systems, (ii) extract information from the execution of the software systems, (iii) select, compose, and reuse existing codes written in K for rapidly constructing the software system, and (iv) perform validation of the target system. The OSAM*.KBMS is represented by the control layer, which includes the facilities for persistent and transient object management, query processing, transaction management, and rule processing. Finally, the control layer is mapped to the storage layer, which includes the facilities for physical storage management (e.g., access methods, file management, and data organization) and low-level transaction management (e.g., concurrency control and recovery).

Note that the OSAM*.KBMS is used not only to model, process, and manage data of application world like the past and present DBMSs, but also to model, process, and manage all software systems as well as their related meta information. All are modeled by the same knowledge model as object classes and processed under the control of the underlying computation model of the KBMS. It is not necessary to make traditional distinctions among software systems (e.g., application systems, operating systems, and DBMS) because all of them are object classes of the underlying universal object-oriented knowledge base as shown in Figure 1.2. The structural and

behavioral properties of all object classes that model programs and application objects can be shared and reused among the users.

A prototype version of K (K.1) and its supporting OSAM*.KBMS have been implemented on Sun 4 in C++ as a first step toward the KBMS-supported software development system described above. This dissertation presents the design and implementation of K in terms of its (i) underlying knowledge model (abstraction layer), (ii) linguistic facilities (language layer), (iii) implementation architecture (control layer), and (iv) application to evolutionary prototyping of software systems. In general, the contribution of this research lies in integrating the techniques introduced in data-base management system, programming language, and software engineering in an object-oriented framework toward a KBMS-supported software development system. Specifically, my contribution to this research is two-fold. First, on the language aspect, I have designed the knowledge-base programming language K as the high-level interface of the KBMS-supported software development system. A detailed description of K will be given in this dissertation. I have also participated in the implementation of the first prototype version of K on Sun 4. An overview of the system architecture and implementation strategy will be given in this dissertation, and a detailed description can be found in Arroyo [ARR92]. Second, on the software development

methodology aspect, I have extended the OSAM* model [SU86,89a, YAS91] with control associations so that both structural and behavioral properties of objects can be uniformly modeled, managed, and processed by the KBMS to support a knowledge-base modeling approach to evolutionary prototyping of software systems [SU92]. This extension paves the way for a software development system in which the developer can graphically model, query, and execute any software system at any level of abstraction. A detailed description of the knowledge model and the KBMS-supported evolutionary prototyping approach will be given in this dissertation.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 surveys the related works in the categories of database programming languages, structural and behavioral modeling, and KBMS-supported software development systems. In Chapter 3, we give an overview of the knowledge-base programming language K in terms of the underlying knowledge model, type system, persistence, and KBMS operations. Structural abstraction mechanisms of K are described in detail in Chapter 4 in terms of the structural association definition, association pattern, and the knowledge retrieval facilities of K. Behavioral abstraction mechanisms of K are described in detail in Chapter 5 in terms of methods and rules. An extended object-oriented computation model for

supporting multi-paradigm computations is given in Chapter 6. In Chapter 7, we present the KBMS-supported evolutionary prototyping methodology. The system architecture and the current implementation strategies are given in Chapter 8. Finally, Chapter 9 gives our conclusion and outlines the future research directions. A syntax summary of K in BNF form is given in Appendix A. A parts knowledge-base example is given in Appendix B to illustrate the expressiveness of K as suggested in Atkinson and Buneman [ATK87].

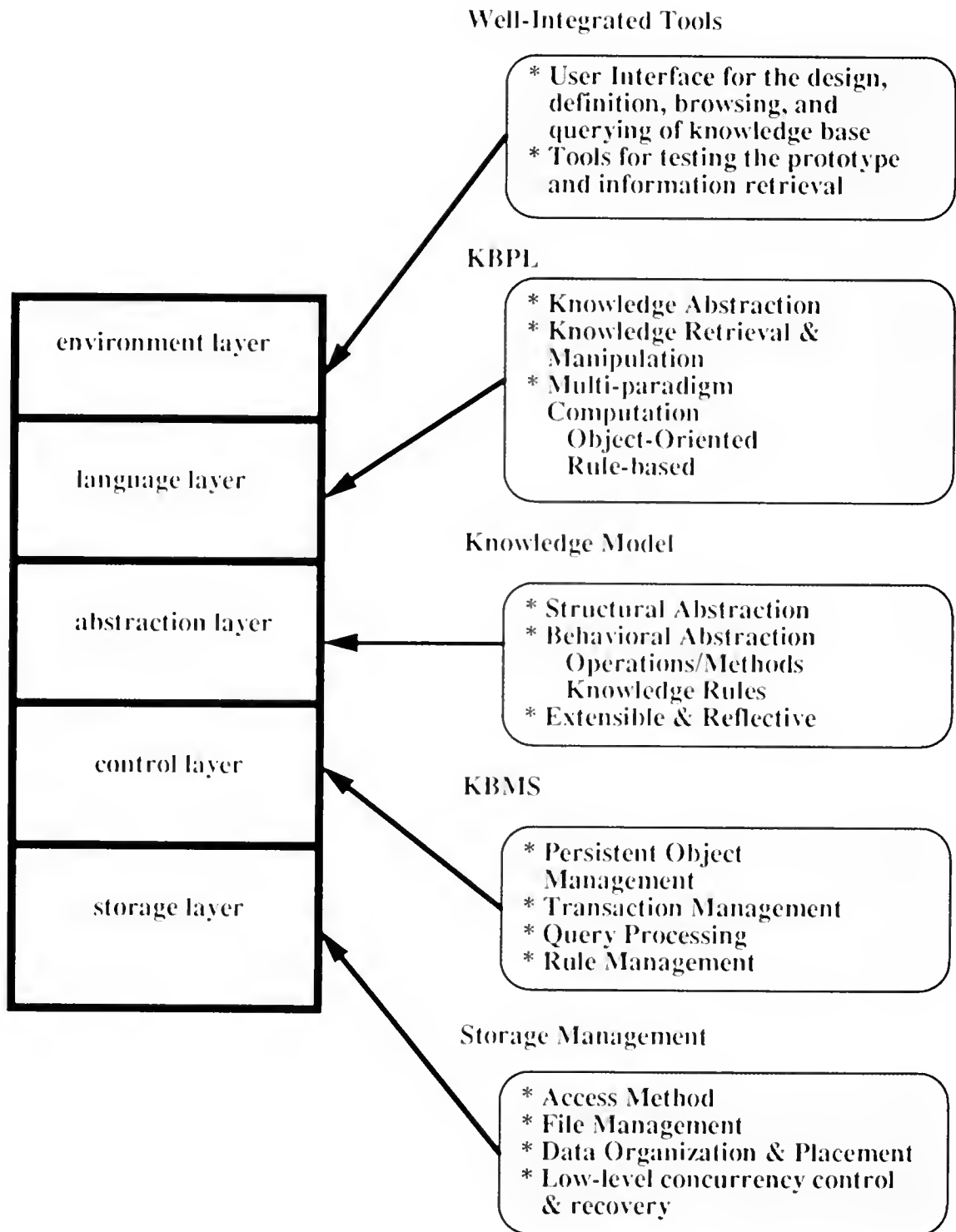


Figure 1.1 The Layer Structure of a Future KBMS-supported Software Development System

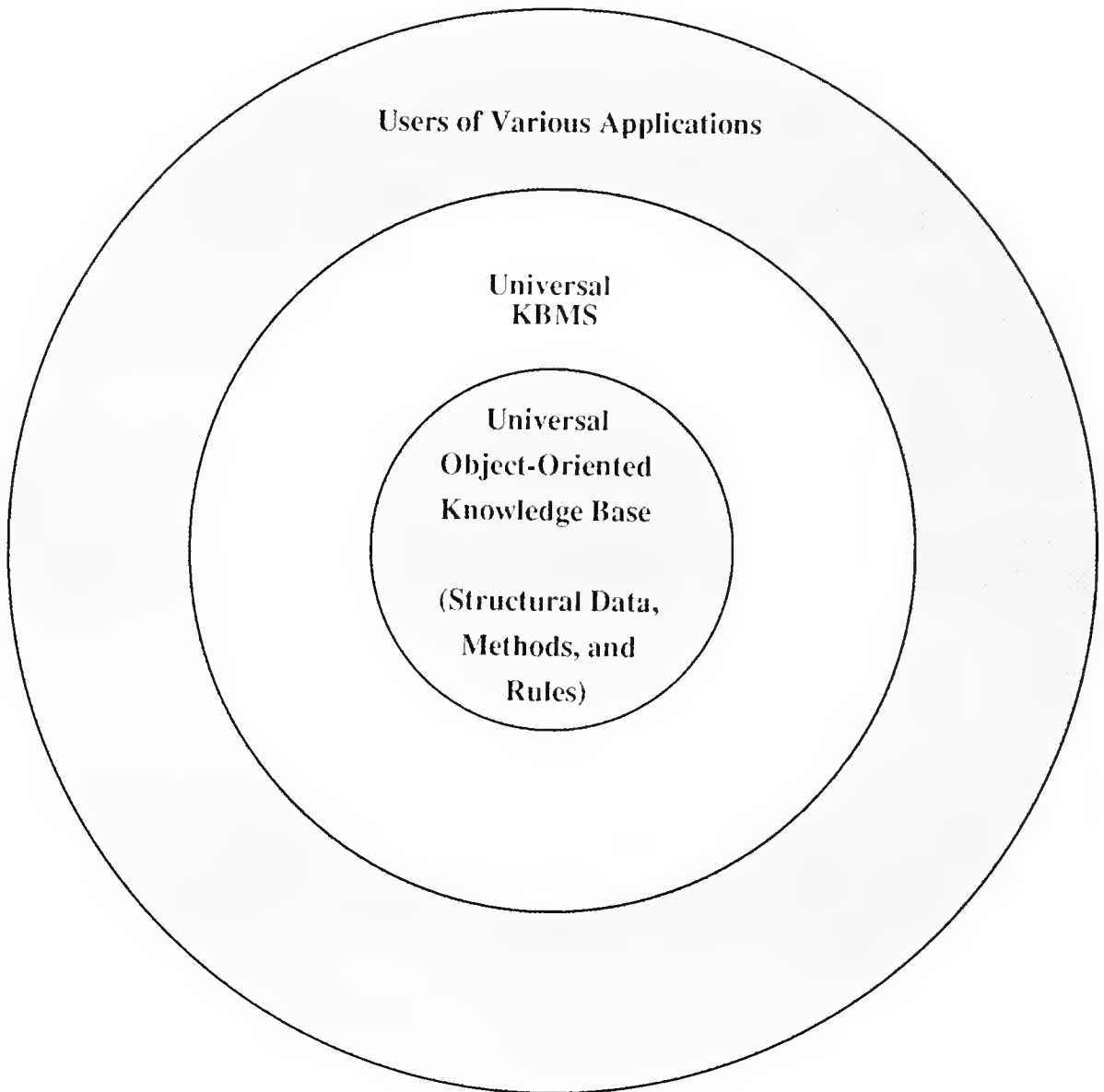


Figure 1.2 A Universal KBMS-Supported Software Development System

CHAPTER 2 SURVEY OF RELATED WORKS

2.1 Data-base Programming Language

Many "data-base programming languages" [ATK87, BLO87] have been proposed in recent years (e.g., Pascal/R [SCH77], Rigel [ROW79], Taxis [MYL80], Dial [HAM80], Plain [WAS81], Adaplex [SMI83], PS-Algol [ATK83], GemStone [COP84, MAI86, BUT91], Galileo [ALB85], Trellis/Owel [SCH86], Vbase [AND87], E [RIC87], Orion [KIM88], Proquel [LIN88], O++ [AGR89], OQL[X] [BLA90], Ontos [ONT91], IRIS [FIS87, WIL90b, ANN91], ObjectStore [LAM91], and O2 [LEC89, DEU91]) to overcome the infamous impedance mismatch problem between traditional programming languages and DDL/DML [COP84, MAI89] by integrating data definition, data manipulation, and general computing facilities in a single language. A detailed survey can be found in Atkinson and Buneman [ATK87].

Most of the existing works are based on relational, functional, or object-oriented data models, with the extension of persistence, the computation facilities of such traditional programming languages as Pascal, Lisp, and C/C++, and associative access (using either iterator or SQL-like construct, both of which can be nested to arbitrary numbers

of levels). They generally do not provide the facility for rule processing which is considered one of the major requirements for next-generation data-base management systems [ATK90, SIG90, SIL91]. While researchers in deductive data-base systems (e.g., LDL [TSU86], LOGRES [CAC90], and Glue-Nail [PHI91]) and active data-base systems (e.g., Postgres [ROW87, STO91], Starburst [LOH91], Ariel [HAN89], HiPAC [CHA89, DAY88], and OSAM*.KBMS [SU89a,b,91, LAM89a,b, CHU90, SIN90]) have tried to extend relational or object-oriented data-base systems with rules, their results in general have provided separate rule languages as an extension of their query languages instead of integrated data-base programming languages. Two existing DBPLs that are most closely related to our work are discussed in the following.

Proquel [LIN88] is the only existing data-base programming language that is specifically designed for the prototyping of data-base applications. With the help of a set of prototyping tools, one can use Proquel to specify, query, and implement relations, events, and operations of an application. Proquel is based on a relational data model and does not properly support complex objects and abstract data types for advanced data-base applications such as CAD/CAM, Office Information Systems, and Software Engineering.

O++ [AGR89, GEH91] extends the traditional object-oriented paradigm with rules and versioning, both of which are useful for the development of complex software systems.

O++ extends C++ with the facilities for creating persistent and versioned objects, defining sets, iterating over sets and clusters of persistent objects, and associating constraints and triggers with object classes.

Our proposed knowledge-base programming language K supports the same object-oriented features found in O++, such as a rich type system, structural associations, operational specifications, rules, object identification, encapsulation, and inheritance. Temporal and versioning facilities will be incorporated in the later version of K. Unlike O++, which is a superset of C++, K is designed to be a high-level programming language. While O++ extends C++ data model with rules, K supports a high-level extensible and reflexive object-oriented semantic association model [SU86,89a, YAS91] where all things, including classes, associations, methods, and rules, are uniformly treated as objects. For example, a user can use the query facility of K to query the meta information from the kernel schema in the same way as he/she can query any application domain. New association types can be defined in K and thus extend the model itself. The implementation of K is also extensible based on an open and modular architecture where each software component is also modeled as an object class. Secondly, while O++ extends the "for" loop construct to iterate over sets, K provides more declarative and concise constructs for specifying queries and rules based on object association patterns. Thirdly, K uses

address-independent object identifiers (soft pointers) as object surrogates rather than using three different types of physical address pointers (persistent, transient, and dual pointers) as in O++. Persistent and transient objects are transparent to the users and are treated in the same way. For example, a query can retrieve both types of objects instead of only persistent objects as in O++. Fourthly, K provides a more flexible type system that supports both static type checking and multiple representations of objects, which is not possible in O++. Lastly, K puts more emphasis on readability and maintainability by providing readable syntax rather than the cryptic and non-intuitive C++ syntax. More specific comparisons will be made throughout this dissertation.

2.2 Structural and Behavioral Modeling

As an extension to relational, semantic, and object-oriented data models, knowledge rules have been incorporated into many research works in next-generation data-base systems such as HiPAC [CHA89], ODE [AGR89], OSAM* [SU89a,b], Postgres [STO91], and Starburst [LOH91]. However, these models do not provide facilities for explicitly modeling method implementations at any level of details.

Object-oriented data model provides a uniform framework by encapsulating both the structural properties and part of the behavioral properties (in terms of signature specifications of methods) of a target system into object

classes. Nevertheless, the implementation part of each method is still left as a blackbox and cannot be further modeled. Because the specification of methods does not carry enough behavioral information, the implementation is often prone to errors. Several research projects have been done in an effort to provide an integrated diagram notation for static and dynamic aspects of software systems. Both Kung [KUN89] and Markowitz [MAR90] tried to combine ER data-model and data-flow oriented process specification as a single graphic design tool for conceptual modeling. However, they do not explicitly model process implementations. Besides, as behavior properties (processes) are not incorporated into an object-oriented framework, they cannot take advantage of an object-oriented paradigm such as inheritance and object-oriented data-base system support.

Brodie and Ridjanovic [BR083] proposed ACM/PCM (Active and Passive Component Modeling) methodology for structural and behavioral modeling of data-base applications using an integrated object/behavior schema. Three types of control abstractions (sequence/parallel, choice, and repetition) are used to represent the behavioral relationships between an operation and its constituent operations. Since behavioral properties are explicitly modeled only at a gross level of detail by relating operations to form high-level, composite operations, there is not enough information for the behavioral

schema to be executable and evolve into the target system at the implementation level.

Kappel and Schrefl [KAP91] proposed object/behavior diagrams as a uniform graphic representation of object structure and behavior based on a semantic data model and petri nets. Behavior diagrams are split into (i) life-cycle diagrams, which identify possible update operations and their possible execution sequences with synchronization constraints, (ii) activity specification diagrams, which represent method specifications, and (iii) activity realization diagrams, which represent method implementations at any level of details. Though closely related to our work, the object/behavioral diagram is more of a graphic design tool than a formal knowledge model. Because there is no kernel model to model object/behavior diagrams themselves, software systems represented by these diagrams cannot be uniformly modeled and managed by some underlying KBMS. For example, a user will not be able to query a data base about the structural and behavioral properties of objects.

2.3 KBMS-supported Software Development System

The research in KBMS-supported software development is closely related to our work as it incorporates some data-base and knowledge-base technologies to support the development of software systems.

The CHI system [SMI85] and TI project [BAL81,85, PAR83] are knowledge-based programming systems supported by a main-memory knowledge base and a wide-spectrum language called V and GIST, respectively, to express all stages of the program development process. V is used to describe conceptual models of domains, formal requirements and specifications, programs, derivation histories, transformation rules, relationships between objects, and properties of objects. V covers high-level program specification to low-level control constructs as well as programming language knowledge (synthesis rules, synthesis plans, and constraints on programs). V also serves as the query and access language of the knowledge base. The expressive capabilities of GIST include historical reference to past process states, constraints, demons (asynchronous process responding to defined stimuli), and a relational and associative data model. Both compilers of V and GIST are transformational systems and require human assistance to perform complex implementation steps (they are not automatically translatable into efficient code). Besides, both works are supported by some in-memory knowledge base of programming rules instead of a full-fledged and well-integrated KBMS for persistency, secondary storage management, as well as uniform modeling and management of the software development environment.

The DAIDA project [JAR90, CHU91] is also a knowledge-based software development system. It uses three different

languages for the description of the software at each stage: (i) a temporal knowledge representation language--Telos--for requirement specification and domain analysis, (ii) a design language--TDL--for identifying and specifying the data and procedural components of the system, and (iii) a data-base programming language--DBPL--(which is an extension of Modular-2) for implementation. Meta information such as design decisions and rules can also be modeled in Telos, which is supported by a KBMS called ConceptBase. From a software development point of view, there are two problems with DAIDA. First, instead of using a single wide-spectrum language for both the specification and implementation, DAIDA uses different languages at different stages. Second, while Telos and TDL are based on an object-oriented data model, the implementation language DBPL is based on a relational data model and therefore a mapping from object-oriented design is needed.

CHAPTER 3 LANGUAGE OVERVIEW

3.1 Knowledge Abstractions

3.1.1 Classes

We use classes as the knowledge definition facilities to classify objects by their common structural and behavioral properties in an integrated fashion. Classes are categorized as entity classes (E_Class) and domain classes (D_Class). The sole function of a domain class is to form a domain of possible values from which descriptive attributes of objects draw their values. Both primitive domain classes (e.g., integer, real, and string) and complex domain classes (e.g., date and address) are supported in K. An entity class, on the other hand, forms a domain of objects that occur in an application's world and can be physical entities, abstract things, functions, events, processes, and relationships. The structural properties of each object class (called the defining class) and thus its instances are uniformly defined in terms of its structural associations (e.g., aggregation and generalization [SMI77]) with other object classes (called the constituent classes). Each type of structural association represents a set of generic rules that govern the knowledge-

base manipulation operations on the instances of those classes that are defined by the association types. Functional associations between object classes can also be specified by such association types as "friend" [STR86] and "using" [BOO90] to facilitate "programming in the large", as will be described in Chapter 4. Manipulation of the structural properties of an object instance is done through methods, and the execution of methods is automatically governed by rules to maintain the system in a consistent state or to trigger some pre-defined actions when certain conditions become true. In other words, the behavioral properties of each object class are defined as methods and rules applicable to the instances of this class. Since rules applicable to the instances of a class are defined with the class, rules relevant to these instances are naturally distributed and available for use when instances are processed. The procedural information (algorithm) of methods can be explicitly modeled using control associations, which will be described in Chapter 7. Structural associations, functional associations, and control associations are all called "class associations" as each of them models the relationships between the defining class and constituent classes. A schema is defined as a set of class associations.

In general, a class definition consists of association section, method section, rule section, and implementation

section (which contains the actual codes that implement the methods specified in the method section) as follows:

```
entity_class | domain_class <name> is
    [associations: association_definition_statements]
    [methods: method_definition_statements]
    [rules: rule_definition_statements]
    [implementation: method_implementation_statements]
end <name>;
```

Note that in this dissertation, we use square brackets to denote optional occurrence of the enclosed construct, and curly brackets to denote an arbitrary number (possibly zero) of the occurrences of the enclosed construct. A sample entity class definition of Student is given in Figure 3.1 to illustrate the skeleton of a class definition. A detailed description will be given in the latter chapters. Note that while implementation section is separated from specification sections as in abstract data types, it is still physically part of the class definition for the following reasons. First, we want to enforce modularity at the granularity of classes and therefore to reduce the overhead of file management and code generation. In other words, each class definition must reside in exactly one file instead of one or more files as in C++. Second, we want to replace the operating system concept of files by the logic level concept of classes as much as possible and eventually hide file management totally from the

user's point of view. When a complete programming environment is available, all the specification and implementation of a class will be done via a graphic user interface instead of by editing files explicitly, and each class definition will be translated into a corresponding K file internally for compilation.

3.1.2 Objects and Instances

Objects are categorized as domain class objects (D_Class_Object) and entity class objects (E_Class_Object). Domain class objects are self-named objects which are referred by their values. Entity class objects are system-named objects each of which is given a unique object identifier (oid). We adopt a distributed view of entity class objects to support generalization and inheritance as in Lam and Alashqur [LAM89a] and Yassen et al. [YAS91] by visualizing an instance of class "X" as the representation (or view) of some object in class "X." Each object can be instantiated (as an instance) in different classes with different representations but with the same oid. Each instance is identified by a unique instance identifier (iid), which is the concatenation of cid and oid, where cid is a unique number assigned for each class in the system and is defined as the type of this instance. Given an iid, the system will use its (i) cid part to refer to a particular class, and (ii) oid part to refer to the representation of a particular object in this class. For two

entity classes "A" and "B," if Class "A" is a superclass of class "B" (generalization association), then for each object that has an instance in class "B," it must also have an instance in class "A." Both instances have the same oid and are conceptually connected by a generalization association link. The advantages of the distributed view of objects are four-fold. Firstly, as inheritance is handled by the object manager at the KBMS layer rather than being built into the storage layer, we achieve a higher level of abstraction and better system independence. For example, it is possible to replace the storage layer with a relational data-base engine by modifying the mapping from the KBMS layer to the storage layer without affecting the language layer. It is also possible to extend the knowledge model at the abstraction layer by modifying the mapping from the abstraction layer to the KBMS layer without affecting the storage layer. Secondly, we have more flexibility in supporting multiple representations of object. For example, one object can span more than one generalization hierarchy path. At the implementation level, it is easier to delete an instance without extra copying data and changing addresses. We also have the flexibility to cluster objects either by "oid" or by class. Thirdly, it is more efficient on scan-based selection as the size of each object is significantly reduced by partitioning it into different instances. Fourthly, as the manipulation of objects is done at the instance level, static

type checking can be easily supported. Each entity class is associated with an extension, which is the set of all its instances.

3.1.3 Encapsulation and Inheritance

We adopt the C++ three-level information hiding mechanism [STR86] by classifying aggregation associations (which are expressed as attributes, data members, or instance variables in other object-oriented programming languages to describe the state of an object instance) and methods as either "public," "private," or "protected." Note that all the rules are treated as "protected" by definition. Public properties can be accessed (i.e., visible) in the implementation of any class in the system, while private properties can only be accessed in the implementation of the defining class and any "friend" of the defining class (specified by the "friend" association, which will be described in Chapter 4). Protected properties are similar to private properties except that they can also be accessed in the implementation of any subclass of the defining class (i.e., subclass-visible). At the class level, all the (i) public and protected aggregations, (ii) public and protected methods, and (iii) rules defined by a class are inherited by its subclasses. At the instance level, an instance of entity class "A" stores only the attributes defined for "A," and it inherits (i.e., gets access to) all the public and protected attributes from its corresponding

instances (with the same oid) of all the superclasses of "A." Name conflict in multiple inheritance is resolved by requiring the user to (i) specify from which superclass a particular property is inherited, or (ii) cast the type (i.e., the cid part of an iid) of an instance to that specified superclass to refer to the corresponding instance explicitly as will be discussed in Section 3.4.

3.2 Model Extensibility and Reflexivity

Model extensibility is achieved via a reflexive kernel model shown in Figure 3.2 in which all the data model constructs described above such as classes, associations, methods, and rules are modeled as first-class objects. One can extend the data model by modifying this set of meta classes. This kernel model also serves as the data dictionary as all the object classes in the system are mapped into this class structure. One can therefore browse and query any user-defined schema as well as the dictionary uniformly. Note that the kernel class structure is in turn mapped into a C++ class structure at the implementation level that is hidden from KBCs (Knowledge-Base Customizers, who customize the data model for specific applications), KBAs (Knowledge-Base Administrators, who use the model to model an application domain), and end users. Note that Figure 3.2(a) illustrates the overall generalization lattice, and Figure 3.2(b) shows the detailed structural relationships among those kernel object classes,

as we will describe in the following sections. In our graphic schema notation, (i) entity classes and domain classes are represented as rectangular nodes and circular nodes, respectively, (ii) a generalization association is represented by a "G" link from a superclass to a subclass, and (iii) an aggregation association is represented by an "A" link from the defining class to a constituent class. Note that the root class "Object" is represented by a special notation because it is neither an entity class nor a domain class. The sole function of class "Object" is to serve as the collection of all the objects in the system. After compilation, any user-defined class (e.g., "Person" and "Student" in Figure 3.2(a)) will be added to the class structure as an immediate or non-immediate subclass of either "E_Class_Object" or "D_Class_Object," while at the same time the objects corresponding to the class definition, associations, methods, and rules of the defining class will be created as instances of the system-defined entity classes named "Class," "Association," "Method," and "Rule," respectively. Note that this class structure is reflexive in the sense that we use the model to model itself. For example, while any user-defined or system-defined entity class is a subclass of "E_Class_Object," "E_Class_Object" itself is also an entity class (represented by a rectangular node). Similarly, "D_Class_Object" itself is also a domain class.

As any application domain (including the model itself) is uniformly modeled and mapped into the kernel model, the class structure can be further extended at any level of abstraction. For example, one can use the kernel model to incrementally extend the model itself by either (i) adding new structural association types or introducing subtypes of existing association types (e.g., "Interaction," "Composition," and "Crossproduct" [SU89b]) by specifying their structural properties (in terms of existing structural association types) and behavioral properties (in terms of generic rules that govern the knowledge-base manipulation operations on the instances of those classes defined by the association types) or (ii) extending the definition of existing association types (e.g., add new attributes "default_value," "null_value," "optional," "unchangeable," and "dependent" [SHY91], as well as their corresponding generic rules for the association type "Aggregation") so that more semantics can be captured in the schema and maintained by the KBMS instead of being buried in application codes. Once a new association type is defined, it becomes a semantic construct of the extended data model and can be used in the definition of any object classes (including any other new association type). In such a way, the data model itself can be incrementally extended to meet the requirements of various application domains.

3.3. Persistence

As a data-base programming language, K must support persistence so that objects can live after the execution of a program is terminated. Persistence in K is based on the following rationales:

(1) Persistence is orthogonal to classes as in Exodus/E [RIC87], ODE/C++ [AGR89a,b], ONTOS [ONT90], and OQL[X] [BLA90], i.e., persistence is an instance property rather than a class property. Any subclass of "E_Class_Object" automatically inherits the persistence mechanism and it is up to the user to specify each of its instances to be either persistent or transient using the pnew or new operator when creating a new object, respectively. By specifying an object instance in a particular class to be persistent, we automatically make all the instances of that object to be persistent. A transient entity class object is similar to a dynamically allocated heap object in C++, which exists in the main memory until explicitly deleted or when the program execution is terminated. Domain class objects can be persistently stored in the data base only if they serve as attribute values of some persistent objects. Note that dangling references might exist after program execution is terminated if some transient entity class instances are referred by persistent entity class instances. Following the same philosophy of Richardson and Carey [RIC87], Agrawal and

Gehani [AGR89], and Blakeley [BLA90], K also assumes that it is the user's responsibility to take care of this problem. In the later version of K, we will consider to solve this problem by having the object manager to delete all the transient object instances (and thus all the associations with them from persistent entity class instances) before closing the knowledge base.

We do not further classify entity classes as persistent entity classes and transient entity classes as proposed in Richardson and Carey [RIC87] and Blakeley et al. [BLA90] for the following reasons. Firstly, based on the rationale that persistence is an instance property rather than a class property, it is reasonable to make any distinction only at the instance level rather than at the class level so that persistence can be strictly orthogonal to classes. Secondly, since we already provide the pnew and new operators, the classification of persistent and transient entity classes becomes unnecessary and only causes less flexibility and more management overhead.

(2) Persistence is orthogonal to oid/iid and any physical address. As a high-level programming language, K provides the user with exactly the same view of objects as the conceptual knowledge model. Unlike other C++-based persistent languages mentioned above, which mix the low-level concepts of pointers (main memory and disk address) with logical oids to improve efficiency, K uses (i) oids for entity class objects and iids

for their instances, and (ii) values for domain class objects, no matter whether these objects are transient or persistent. The advantages are three-fold. Firstly, the language provides a better object-oriented flavor and data abstraction than C++-based languages by enabling the users to (i) manipulate objects at the logical level instead of going to the physical level by following pointers, and (ii) navigate through the data-base using oids/iids instead of pointer chasing. Secondly, the concept of multiple representations (views) of objects described in Section 3.1 is well supported at the logic level and clearly separated from the low level implementation. Thirdly, system-defined oid (and iid) is independent of physical address and thus (i) enforces the immutability and uniqueness requirements of identity [KH086] and (ii) makes persistence orthogonal to oids/iids. As a result, unlike in most C++-based persistent languages [AGR89a,b, RIC87,90, BLA90] that put extra burden of managing two or three types of pointers (persistent, transient, and/or dual pointers) on the users, persistence in K is transparent to the user in the way of declaring an entity class instance variable. The reason is that such a variable in K will be bound to iid, which is independent of any physical location or persistence property.

(3) Persistence is orthogonal to queries and any object manipulation. Since persistence is a property that makes difference only after the program execution is terminated, we

feel that there should be no difference in queries and manipulation of persistent and transient objects. For example, a selection query over an entity class should return both its persistent and transient instances as long as they satisfy the selection condition as in Blakeley et al. [BLA90]. There is also no difference in object manipulation because all the entity class objects are manipulated using oid/iid, which is independent of persistence property as described in the second rationale.

The overhead of implementing persistence in K is that for each entity class, a persistent object table "PerTable" (which itself is also persistent) and a transient object table "TransTable" (which itself is also transient) are needed for mapping oids of its persistent and transient object instances to their main memory addresses, respectively. Note that the address of a persistent object instance will be a special value denoted by "INACTIVE" if it is not in the main memory. During run time, the persistent object tables will be copied into the main memory for fast retrieval. If a persistent object instance is needed and its address is "INACTIVE," the object manager will automatically call the ONTOS function "OC_DirectActivateObject" to retrieve it into the main memory. A performance penalty is introduced due to this one-level indirection of object references. Object tables are currently implemented by using the indexed dictionaries to speed up this process [ARR91]. The object manager will be responsible for

maintaining both object tables corresponding to various method calls by updating these tables accordingly.

3.4 Type System

Different from conventional object-oriented programming languages, K directly manipulates objects at the distributed instance level as described in Section 3.1, and instance identity is used as a primitive for type management. In order to support the design principle of static type checking, we incorporate a strong notion of typing into the knowledge model by defining the type of an instance as the class to which this instance belongs. Every variable in K will be bound to some instance and therefore must be declared to have a type. Every expression in K will be evaluated to return an instance and thus also has a type that will be identical to the type of the returned instance. Note that static type checking is supported in K because (i) K directly manipulates instances rather than objects, and (ii) the type of an instance is fixed. Some programming constructs for performing the same effect of late binding as in conventional object-oriented programming languages without sacrificing static type checking will be discussed in Chapter 4.

The type of an expression in general can be detected by the system type checker by textual inspection (static type checking) to decide the type compatibility and thus prevent an operation from being applied to a value of an inappropriate

type. Type compatibility means that a variable of type X can only be assigned expressions that represent instances of class X or any subclass of X. For the later case, the system will automatically convert the type of the instance, which is returned by the right-hand-side expression to class X during run time, to actually refer it as an instance of class X. Method parameters and returned values are checked against the method signature following the same rule as above. An instance variable ranging over an entity class will be bound to some iid from which the system uses its (i) cid part to refer a particular class (which is the type of this instance), and (ii) oid part to refer the representation of the particular object in this class. Conceptually, an entity class object can dynamically gain/lose types during program execution when its corresponding instances are created/deleted as we will illustrate in Section 3.5. An instance variable ranging over a domain class will be bound to some value.

If the type checker is not able to ascribe a type to an expression, the user must use the cast operator '\$' to specify the type in the form "<class>\$<expression>." The cast operator is useful for the user to temporarily convert the type of an expression or refer different representations of the same entity class object in different classes. For example, `real$(3+4)` asserts that the type of (3+4) is real instead of integer. Similarly, to resolve any name conflict in multiple inheritance, one must specify from which superclass a

particular property is inherited by casting the type of an expression to that specified superclass to refer the corresponding instance explicitly. For example, if both classes Student and Employee define a method called "evaluate," and both classes have class TA as a subclass. To apply "evaluate" to a TA instance "t," one must use either "Student\$t.evaluate()" or "Employee\$t.evaluate()" for the system to unambiguously apply the correct method to corresponding Student instance or Employee instance of "t," respectively. Note that in the cases when no name conflict occurs, the system will automatically find the appropriate superclass and perform the casting to support inheritance. In other words, inheritance at run time is supported by casting an instance of class "X" to be an instance of class "Y" (which is a superclass of class "X") before accessing a property defined by class "Y."

For primitive domain classes, type conversion can only be done (i) from character to integer or string, (ii) from integer to real, and (iii) from real to integer. For complex domain classes, only upward conversion is allowed, i.e., a value from domain class X can be converted to a value of any superclass of X, but not vice versa. On the other hand, there is no restriction for type conversion of entity class instances because we allow an entity class object to have different representations in different classes. The advantage of this approach is more flexibility in modeling an

application domain, especially for the purpose of prototyping. For example, before class Robot is defined (because its properties are not finalized), one can still create an object as both an instance of Person and an instance of Machine to prototype its structural and behavioral properties.

Note that while upward casting to superclass is always safe (an instance in class X guarantees a corresponding instance with the same oid in every superclass of X because of generalization association), null value might be returned in other cases if there does not exist any corresponding instance in the target class. In other words, one can use the cast operator to test if an instance of class X also has a corresponding instance in class Y. Also note that we do not allow the use of casting to bypass and thus corrupt the information hiding mechanism described in Section 3.1. In other words, if property P of class A is not visible in class B, then the expression A\$x.P (where x is an entity class instance variable) used in the implementation of class B will be detected by the compiler as a semantic error.

3.5 KBMS Operations

Entity class objects are directly manipulated at the instance level in K. After an entity class is defined, we can insert instances into this class. An instance can be created from scratch by using the "new" or "pnew" operators followed by <class_name>(<attribute_assignments>) to create a new

transient or persistent object along with an instance of this object in the specified class, respectively. Figure 3.3 illustrates the basic KBMS operations and object/instance concept using a simple schema where class "Student" is a subclass of "person" and a superclass of both "TA" and "RA."

Statements (1) and (2) first create two new objects with their oids, insert their instances in class Person and class Student, and return the iids to variables "p" and "s1," respectively. Note that by inserting a Student instance, the system object manager automatically inserts a corresponding Person instance (with the same oid) because of the generalization association. The difference between the instances referred by "s1" and "p" is that the former is a persistent instance and therefore any modification will be written to the data-base while the latter is a transient instance that resides only in main memory. Note that while primitive domain class instances (e.g., integer and string) can be directly referred by their values, complex domain class instances (e.g., date) can be referred in the form "<class_name>(<attribute_assignments>)," as shown in statement (1). The creation of entity class object instances can also be nested within the creation of other entity class object instances as shown in statement (2). Statement (3) then inserts a new TA instance of the object referred by variable "p" (assume we learn that this person is a TA) and returns the iid to variable t. Note that (i) no new object is created and

the iid returned to t has the same oid as "p," and (ii) by inserting a TA instance of the object referred by "p," the system object manager automatically inserts a Student instance of the same object. Similarly, statement (4) inserts another instance of the same object referred by "p" in class RA and returns the iid to variable "r." Statement (5) casts the TA instance referred by "t" as a Student instance whose iid is then assigned to "s2." In statement (6), we delete the TA instance "t" and the object manager will automatically delete all the instances of the object referred by "t" in all the subclasses of TA (if any) following the generalization association. This can be done by (i) identifying all the immediate subclasses of TA that contains an instance of the specified object, and (ii) performing the delete operation recursively on the corresponding instances from these subclasses. All the references (association links) to these deleted instances from other object instances will be automatically removed to maintain the referential integrity constraint. Note that for this particular object, even though it lost its instance in TA, it still has its instances in RA, Student, and Person. For example, we can use the variable "s2" from statement (5) to refer to its Student instance. In other words, we allow an object to have different representations in different classes (even spans more than one branch of the generalization lattice), and we can insert or delete these representations dynamically. This is a property that cannot

be expressed in most object-oriented data-base programming languages except those systems that support multiple views of objects (e.g., Aspect [RIC91], IRIS [FIS87, WIL90a, ANN91], and Clovers [STE89]). Note that we achieve this flexibility without losing the advantages of static type checking by (i) explicitly making the distinction between objects and instances and (ii) directly manipulating instances rather than objects. In statement (7), a destroy statement will automatically delete all the instances of the object referred by variable "p." Note that it is impossible to restrict an object to have instances along only one branch of the generalization lattice when multiple inheritance is allowed. Therefore, a "destroy" operation must be done by performing a "delete" operation on the corresponding instance from the root class E_Class_Object. A prototype implementation of the object manager has been reported in Arroyo [ARR92].

```

entity class Student is
  associations:
    specialization of Person; /* Student is a subclass of Person */
    friend of Faculty; /* authorize Faculty to access the private and protected properties */
    aggregation of
      public: /* definition of public attributes */
        enroll: set of Course; /* a student can enroll in a set of courses */
        college_report: array [4] of GPA_Value; /* annual report of every college year */
        major: Department;
      protected: /* definition of protected attributes */
        S#: S#_Value;

  methods: /* the signature of methods */
    public:
      method eval_GPA() : GPA_Value;
    private:
      method suspend() : void; /* no return value */
      method inform_all_instructor() : void;

  rules:
    rule CS_rule1 is
      /* after updating the major of a student, if the new major is "CIS"
         then the GPA of this student must be greater than 3.0,
         otherwise we suspend this student */
      triggered after update major
      condition (this.major.name = "CIS" | this.eval_GPA() > 3.0) /* guarded condition */
      otherwise this.suspend()
    end CS_rule1;

    rule Student::General_rule1 is
      /* after suspending a student, if this student enrolls in any course,
         then inform all the instructors of this student */
      triggered after suspend()
      condition exist c in this *>[enroll] c:Course /* existential quantifier */
      action this.inform_all_instructor()
    end General_rule1;

  implementations: /* actual coding of methods */
    method eval_GPA() : GPA_Value is
      local s1, s2 : real := 0; GPA : GPA_Value; /* local variable declarations */
      begin
        context this *<[student] t:Transcript *>[course] c:Course /* looping over a context */
        do begin /* for each <this,t,c> tuple, do the following */
          s1 := s1 + c.credits * t.grade_point; /* calculate the accumulated grade points */
          s2 := s2 + c.credits; /* calculate the accumulated credit hours */
        end;
        end context; /* end of context looping */
        GPA := s1/s2;
        if GPA < 2.0
          then "GPA Below 2.0".display();
        end if;
        return GPA;
      end eval_GPA;
end Student;

```

Figure 3.1 The Class Definition of Entity Class Student in K

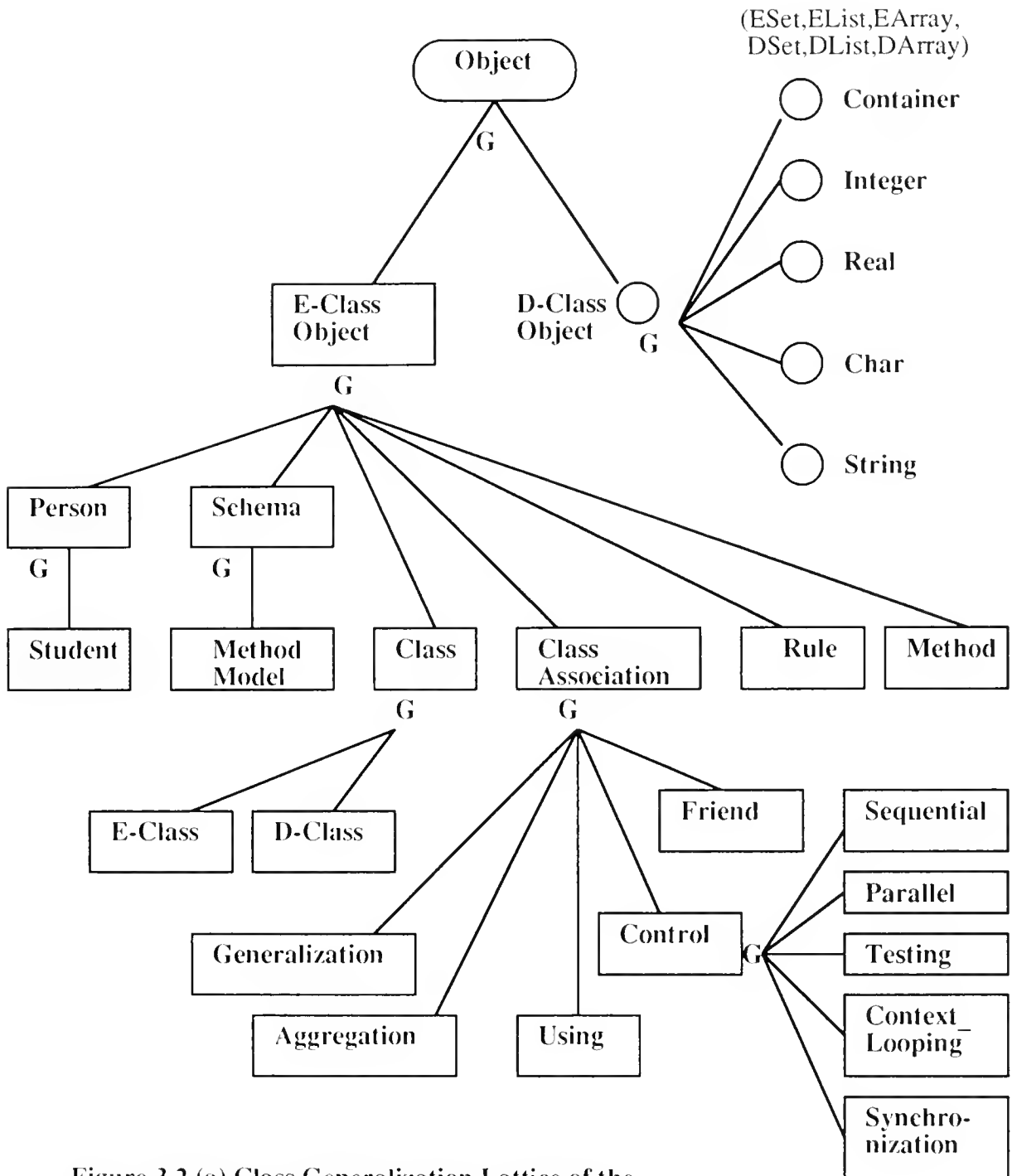


Figure 3.2 (a) Class Generalization Lattice of the Extensible Kernel Model

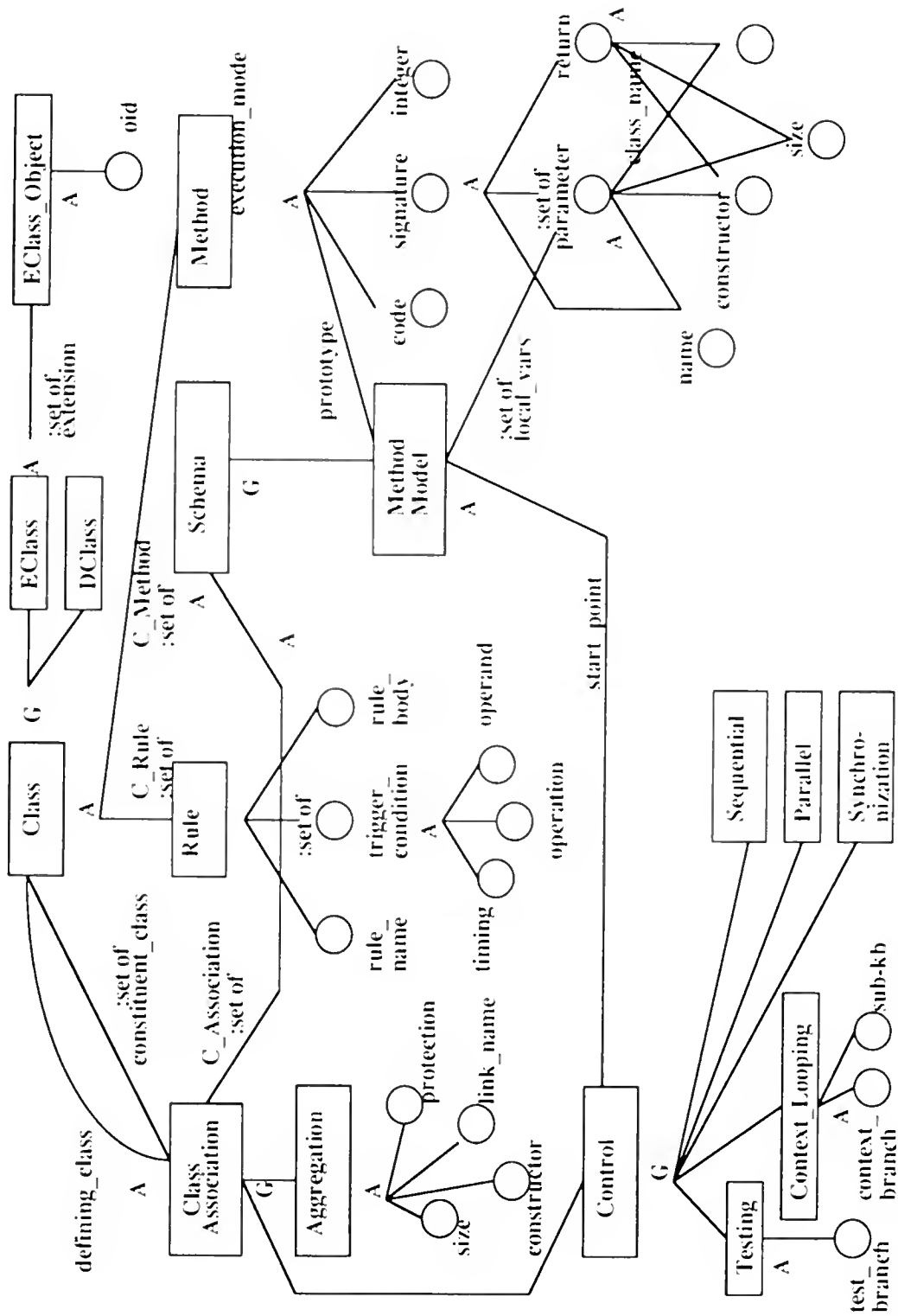


Figure 3.2 (b) Aggregation Structure of the Extensible Kernel Model

```

local p:Person; s1,s2:Student; t:TA; r:RA;
begin
  (1) p := new Person(name := "Anne Rice",
                      birth := date(month := 1,
                                   day := 1,
                                   year := 1950));
  (2) s1 := pnew Student(s# := "1234",
                        name := "Steven King",
                        guardian := pnew Person
                                (name := "James Michener"));
  (3) t := p insert TA (Course := "CIS6501");
  (4) r := p insert RA (project := "OODB");
  (5) s2 := Student$t; /* casting t to student */
  (6) delete t;
  (7) destroy p;
end;

```

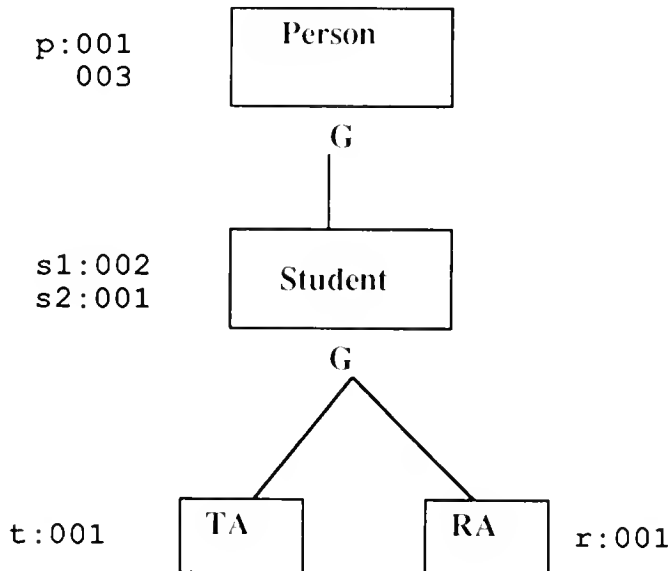


Figure 3.3 KBMS Operations Example in K

CHAPTER 4

STRUCTURAL ABSTRACTION MECHANISMS

It has been shown in Rumbaugh [RUM87] and Wile [WIL90a] that Structural associations (or relationships) serve as an important abstraction mechanism, which is missing in traditional programming languages. In our work, structural properties of objects are modeled by various structural association types, which are uniformly modeled as first-class object classes as shown in Figure 3.2. Note that while conceptually, each type of structural association represents a set of generic rules that govern the knowledge-base manipulation operations on the instances of those classes that are defined by the association types, kernel structural association types are actually built into the object manager for bootstrapping and better performance [LAW91, YAS91, ARR92]. In the later version of K that supports generic rules, we will also allow the user to use the linguistic facilities to incrementally extend the knowledge model as will be described in Chapter 9. We also provide the constructs for specifying object association patterns based on which the system can identify the corresponding sub-knowledge-bases that satisfy these intensional patterns. Two applications of association patterns, namely, a context looping construct for

manipulating the knowledge base, and existential/universal quantifiers for posing logical questions upon the knowledge base, are also described in this chapter.

4.1 Association Definition

As shown in Figure 3.1, associations are defined following the key word "associations:" of a class definition. Two kernel structural association types "generalization" and "aggregation" are currently built into the object manager of OSAM*.KBMS and supported in K.

4.1.1 Aggregation Association

For each object class, one can define a set of attributes to describe the state of its instances in terms of their associations with other classes by using the aggregation (A) association type. The syntax for the Aggregation association definition is the following.

```
aggregation of
public: | private: | protected:
    <aggr_specifications>
{public: | private: | protected:
    <aggr_specifications>
}
```

where <aggr_specifications> is a list of specifications of the form: "<name>:[<collection> of] <constituent_class>;" Each aggregation specification corresponds to an instance of the class "Aggregation" and also a named A-link from the

defining class to the constituent class in the structural schema as we described in Section 3.2. The name of an attribute must be unique within the defining class. The key words public:, private:, and protected: are used to specify different levels of information hiding described in Section 3.1. An aggregation association defines either (i) a value attribute if its constituent_class is a domain class, or (ii) a reference attribute if its constituent_class is an entity class. At the instance level, we store values and iids for value attributes and reference attributes, respectively. Multi-valued attributes are specified as "<collection> of <constituent_class>" where <collection> could be either "set," "list," or "array [<size>]." Note that in addition to the constructor "set," which is critical in object-oriented database systems, we also provide the constructors "list" and "array" to capture the semantics of "order," which is useful in real-world applications [ATK90, SIG90]. The size of an array is specified by an integer, and the index of the first element of an array is always 1. Also note that collection constructors are implemented as domain classes as shown in Figure 3.2(a). Each instance of a collection class corresponds to a collection of iids or values. The semantic type checker of K will use the information carried by each iid to ensure the type compatibility of each collection operation. Note that an aggregation association between two entity classes is interpreted as a bi-directional link. For example, suppose

there is an aggregation association called "major" from entity class "Student" to entity class "Department" having a cardinality mapping of n-1, then the system will automatically define and maintain a set-valued aggregation association called "_KINVStudent_major" from "Department" to "Student" (for each department, the system records all the students who major in this particular department). The system will use this information to support bi-directional navigation and to maintain the referential integrity of the knowledge base. For example, before deleting a Department instance, the system can follow its "_KINVStudent_major" links to identify those students who major in this department and remove their "major" links to this particular Department instance. To access an attribute value or a particular element from a list- or array-valued attribute, we use the conventional dot notation `<instance>.<attribute>` (e.g., "s.major") or `<instance>.<attribute>['<index>']` (e.g., "s.college_report[1]"), respectively.

4.1.2 Generalization Association

For each object class, one can use generalization (G) association to specify its immediate superclass or subclass. The syntax for a Generalization association definition is as follows:

generalization | specialization of <constituent_classes>;

Class "A" is said to be a superclass of class "B" (i.e., there is a generalization association from "A" to "B") if for each object that has an instance in class "B," it also has an instance in class "A." Both instances have the same oid and are conceptually connected by a G-link. Note that generalization association is bi-directional and can be specified in either direction. For example, to say "Student is a specialization or subclass of Person" is equivalent to say "Person is a generalization or superclass of Student." In general, specialization is used to construct object classes in a top-down and step-wise refinement approach by giving more and more structural and behavioral properties. Contrary to most existing object-oriented models, we also allow the user to define specializations of primitive system-defined domain-classes (e.g., a subset of integer) by using some constraint rules to specify range, enumeration, or any other constraints.

4.1.3 Friend and Using Associations

As each class can be thought of as a reusable software module in object-oriented software development, two types of functional associations are provided in K to facilitate "programming in the large." Functional associations can be defined in the form of "<association> of <constituent_class> {, <constituent_class>}," where <association> could be either friend or using. The introduction of the "friend" and "using" associations also illustrate the extensibility of the

knowledge model. In the later version of K that supports generic rules, we shall allow the user to incrementally extend the knowledge model as described in Section 3.2.

Friend. This association type is used to support the three-level information hiding mechanism described in Section 3.1. A "friend" (F) association specifies that all the constituent classes are "friends" of the defining class and thus authorizes them to access the private and protected properties of the defining class.

Using. Similar to the "#include" macro in C++, a "using" (U) association specifies that all the public interfaces defined by the constituent classes will be available to the defining class (client-server relationship). Note that though this information has been implicitly captured in parameter specifications and method invocations, we include it at the class level for better readability and maintainability of complex software systems. For example, a user can easily capture the overall structural and functional relationships among system modules by just reading the association definition or graphic display of the system schema rather than going into the detailed codes of each method. Besides, the compiler can make use of the semantic information provided by the "using" associations in a system schema to automatically include all the necessary classes for compilation. Note that the "using" association provides a modular mechanism at a larger granularity than ordinary classes as one can either (i)

functionally compose many classes into a big module structure or (ii) functionally decompose a big module into smaller modules.

4.2 Association Pattern

Since K serves as the high-level interface of OSAM*.KBMS, the development and execution of a K program would generally involve the processing of a persistent knowledge base. For knowledge-base retrieval and manipulation, a knowledge-base programming language should include some knowledge manipulation constructs in addition to general programming constructs. In our work on K [SHY91], we use pattern-based querying constructs for this purpose. We modify the context expression of OQL [ALA89a,b, GU091] as the primitive construct for specifying structural association patterns based on which the system can identify the corresponding contexts that satisfy the intensional patterns. In the following, we will describe various forms of association pattern in more detail.

4.2.1 Class Expression

The most simple form of association patterns is called class expression, each of which specifies a single object instance or a collection of homogenous object instances (i.e., instances of the same class) of a particular class. A single object instance is represented by an instance variable such as the pseudo variable "this" (which denotes the receiver of

a coming message as in C++) or other user-defined local instance variable. A collection can be specified in one of the following forms: (i) `<E_Class>`, which is the name of an entity class and represents the extension (the set of all persistent and transient instances) of this entity class, (iii) `<E_Class> '[' <boolean_exp> ']'`, which selects instances from an entity class by using `<boolean_exp>` as the intra-class selection condition, and (ii) `<collection_var>`, which is a user-defined variable ranging over an entity class using the constructor set, list, or array. For example, `"s1 : set of Student"` defines a variable "s1" whose value will be a set of Student instances. Note that both implicit sets (class extensions) and explicit sets (user-defined sets) are supported in K. While implicit sets are automatically managed by the system, explicit sets can be manipulated by using set operators "union" ('+'), "intersection" ('&'), and "difference" ('-'). The two operands of a set operator must be defined over the same entity class or domain class. We also overload the '+' and '-' operators to represent the "add" and "remove" operators, which can be used to add and remove a single instance to and from a set, respectively. For example, the statement `"s1 := Student[age > 40] + this;"` adds a student instance denoted by "this" to the set of all students with age greater than 40 and assigns the new set to variable "s1."

One can define a range variable over a collection of instances in a class expression as `"<var>:<collection>"` for

specifying context looping and quantifiers as will be describe in the latter sections. For example, "s:Student[s.eval_GPA() > 3.5]" specifies a set of Student instances, each of which is denoted by the variable "s" and has GPA greater than 3.5. Note that while OQL [ALA89a,b] only allow simple comparisons in a selection condition, K provides more expressiveness and consistence by allowing any boolean expression expressible in K as a selection condition to comply with the orthogonality design principle.

4.2.2 Linear Association Pattern

A linear association pattern is specified as "<class> {<link> <class>}" where <class> represents a class expression described in Section 4.2.1, and <link> is specified as <op><direction>'['<association-name>']'." Note that (i) <op> could be either an "associate" ("*") or a "non-associate" ("!") operator as in Alashgur and Lam [ALA89] and Guo and Lam [GUO91], (ii) <direction> could be either ">" or "<" so that the defining class of <association-name> is always at the open side, i.e., the left-hand-side of ">" or the right-hand-side of "<", and (iii) <association-name> could be either "G" (which represents an generalization association) or the name of an aggregation association. Note that the non-associate ("!") operator is useful in identifying instances at either side of the operator, which are not connected to any instance at the other side. For example, "Student !> [advisor]"

Professor" retrieves not only students who have no advisor but also professors who are not advising any student. While the former can be easily expressed as "Student[advisor = null]"), the later cannot be concisely expressed without the use of "!" operator or quantifiers, which will be described in Section 4.4.

As an example, "g:Grad[major.name = "CIS"] *>[advisor] p:Professor !< [instructor] Course" specifies a sub-knowledge-base that contains (i) all the graduate students of CIS department who has an advisor (i.e., there is an "advisor" link connecting this student with a professor) who does not teach any course (i.e., this professor is not connected through the "instructor" association with any course instance), (ii) all the professors who are the advisors of some students but do not teach any course, and (iii) all the courses that are not taught by those professors who are the advisors of some students. Here, "g" and "p" are variables that represent the graduate students and professors satisfying the association pattern specification, respectively. Note that each course selected in (iii) might be taught by some professor who is not the advisor of any student. A context can be thought of as a normalized relation whose columns are defined over the participating classes and each of its tuples represents an extensional pattern of iids that satisfy the intensional pattern.

The evaluation of the above pattern is illustrated step by step in Figure 4.1. Figure 4.1(a) shows the original knowledge-base. Figure 4.1(b) shows the sub-knowledge-base after we first evaluated "Student *>[advisor] Professor." Only student instances and professor instances, both of which are connected by "advisor" links, are retained. Professor "p3" is dropped. Note that when an instance is dropped, all the links connected to this instance are also dropped. Figure 4.1(c) shows the sub-knowledge-base after we applied the "!" operator. Only those (i) professor instances selected in Figure 4.1(b) that are not connected by the "instructor" link of "Course" with any course instance, and (ii) course instances that are not connected by the "instructor" link of "Course" with any professor instance selected in Figure 4.1(b) are retained. Professor "p1" and course "c1" are dropped. Because professor "p1" is dropped, student "s1" no longer connects to any professor and therefore must also be dropped. Note that conceptually, there is a "non-link" (represented by dotted line in Figure 4.1(c)) connecting each pair of professor and course instances that are not connected to each other by the "instructor" association of "Course". In Figure 4.1(d), we normalize the resulting sub-knowledge-base to get all the combinations of possible <Student, Professor, Course> instances bindings. A sub-knowledge-base is empty if all the columns of its normalized relation are empty. Each tuple of the normalized relation is a set of bindings, which can be

thought of as an extensional association pattern that satisfies the intensional association pattern. Note that the normalized relation as shown in Figure 4.1(d) serves only as a data structure to temporarily store the sub-knowledge-base, and the semantics of each of its tuples (e.g., the "associate" or "non-associate" association between object instances) is interpreted by the system based on the corresponding intensional association pattern.

The use of aggregation association must comply with the information hiding principle described in Section 3.1. An association pattern "A *>[p] B" used in class "X" is valid if and only if one of the following is true: (i) "p" is a public aggregation association defined by class "A" or any of its superclasses, (ii) "p" is a protected aggregation association defined by class "A" or any of its superclasses, and class "X" is either the defining class, a subclass of the defining class, or a friend class of the above classes, and (iii) "p" is a private aggregation association defined by class "A," and "X" is either the defining class "A" or a friend of class "A."

The semantics of generalization association specified in an association pattern is slightly different from that of aggregation associations. As mentioned in Section 3.1, if class "A" is a superclass of class "B", then for each object that has an instance in class "B," it must also have an instance in class "A" and these two instances are implicitly

connected via a generalization link. For example, to find those TAs who are also RAs, we can use "TA *<[G] Student *>[G] RA", which specifies a sub-knowledge-base consisting of all the <TA, Student, RA> triples such that each triple represents three different instances of the same object. In other words, we retrieve all the student instances who are both TA and RA along with their TA and RA instances. Note that we use "<[G]" between TA and Student because TA is a specialization of Student, i.e., Student is a generalization of TA. Similarly, we use ">[G]" between Student and RA because Student is a generalization of RA. Also note that based on our distributed object storage mechanism, the class lattice is not closed under intersection. For example, suppose there is a class called "TA&RA", which is a subclass of both TA and RA. Even though the presence of a TA&RA instance guarantees the presence of a corresponding TA instance and a RA instance, the vice versa is not always true. For example, one may insert an object to class TA and class RA without having to insert this object to class TA&RA. In other words, the intersection of TA and RA is not always equal to TA&RA. The use of generalization association in an association pattern is useful for identifying different instances of an object in different classes to support static type checking as will be described in Section 4.3.

4.2.3 Direction Specification in Association Patterns

Different from OQL [ALA89], link direction in K is explicitly specified for bi-directional navigation to comply with the design principle of readability and maintainability. By following the direction, a user or the system can unambiguously identify which side is the defining class of the association. For the above example, we use the ">" in "Student *>[advisor] Professor" because "advisor" is an association defined by class "Student." Similarly, we use the "<" in "Professor !<[instructor] Course" because "instructor" is an association defined by class "Course." We illustrate the importance of explicit direction by the following two examples. Firstly, suppose class Person defines an aggregation association called "father" whose constituent class is Person itself. The association pattern "this *>[father] Person" specifies the sub-knowledge-base consisting of a particular person instance denoted by "this" as well as the father of this person. Similarly, the association pattern "this *<[father] p:Person" specifies the sub-knowledge-base consisting of this particular person instance as well as all his children (person whose father is this particular person). It is ambiguous if no direction is explicitly given. Secondly, suppose both class Student and class Course define a set-valued aggregation association called "enroll" from Student to Course and from Course to Student, respectively. Similar

to the above example, direction must be explicitly given in "this *>[enroll] Course" and "this *<[enroll] Course" because their semantics are different. Note that the explicit specification of direction allows the user to switch the order of class expressions at the two sides of an association or non-associate operator by just changing the direction. For example, "Student *>[enroll] Course" and "Course *<[enroll] Student" are equivalent in semantics.

From software engineering point of view, relying on the system to infer the direction is also unacceptable for the following two reasons. First, readability is decreased as a reader of a K program must infer the direction by himself. Secondly, maintainability is decreased because the semantics of a valid association pattern would change unexpectedly as the schema evolves gradually. Following the above example, suppose a programmer uses "this *[enroll] Course" to capture the semantics of "this *>[enroll] Course." Later on, the schema is changed so that the "enroll" association from Student to Course is deleted. Instead of reporting a semantic error as it should, the system would automatically interpret this pattern as "this *<[enroll] Course" and thus make the application code difficult to read and maintain. Comparatively, the use of explicit direction produces more readable and reliable code.

4.2.4 Precedence in Association Pattern

Both the association ("*") and non-associate ("!") operators are of the same precedence and are evaluated from left to right in K. To change the order of evaluation, one can use parenthesis "(<pattern>)." Note that changing the order of evaluation will change the semantics of an association pattern only in the case that we want to first evaluate an association pattern at the right-hand-side of a "!" operator before we apply the "!" operator. For example, the following two association patterns (i) "Student *>[advisor] Professor !<[instructor] Course" shown in Figure 4.1 and (ii) "Student *>[advisor] (Professor !<[instructor] Course)" shown in Figure 4.2 are different in the sense that while the former contains all the courses, each of which is not taught by any professor who is also an advisor of some student (but maybe taught by some professor who is not an advisor of any student), the latter contains all the courses, each of which is simply not taught by any professor. In other words, the set of courses selected in (ii) is a subset of that selected in (i). Note that because association operators are of the same precedence and links can be specified in both directions, the association pattern specified in Figure 4.2 can also be expressed as "Course !>[instructor] Professor *<[advisor] Student" without using any parentheses. Here, the evaluation of "Professor

*<[advisor] Student" will not effect the Course instances that have been selected first.

4.2.5 Branching Association Pattern

Tree-structured complex association pattern can be specified using the branch operators "and" and "or" in the form "<class> <branch> (<link><pattern>, <link><pattern>, ...)." The left-hand class expression of a branch operator is called the fork class expression. For the "and" operator, the set of instances returned from the fork class is associated (or non-associated, depending on each <link> specification) with every (or at least one in the case of the "or" operator) pattern on the right hand side of the "and" operator. For example, "Student and (*>[major] Department, !>[advisor] Professor)" specifies a sub-knowledge-base consisting of (i) all the students each of whom has a major and does not have an advisor, (ii) all the departments that these students majoring in, and (iii) all the professors each of whom is not advising any student selected in (i). The evaluation is shown in Figure 4.3. Two sub-knowledge-bases specified by "Student *>[major] Department" and "Student !>[advisor] Professor" are first evaluated separately in Figure 4.3(b). The effect of the "and" operator is to remove those student instances that appear in only one of the sub-knowledge-bases as shown in Figure 4.3(c). In Figure 4.3(d), we represent the resulting sub-knowledge-base as a normalized relation, which can be

thought of as the concatenation of the two previous sub-knowledge-bases in the sense that each tuple is either `<Student, Department, null>` or `<Student, null, Professor>`. Note that no department and professor instance will appear in a tuple at the same time because there is no association specified in the pattern between these two classes. In Section 4.3, we will describe how to iterate over the relation for knowledge-base manipulations. Also note that branching can be nested to form complex patterns. For example, `"Student and (*>[receiver_of] Fellowship, *>[G] Grad or (*>[G] TA, *>[G] RA))"` specifies a sub-knowledge-base consisting of (i) all the students each of whom receives any fellowship and at the same time is a graduate TA or RA, (ii) the fellowships that these students receive, and (iii) the TA or RA instances of these selected graduate students.

4.3 Context Looping Statement and Navigation

4.3.1 Context Looping Statement

As mentioned in Chapter 1, one of the design principles of K is to seamlessly incorporate the high-level query facilities into the language constructs for the retrieval and manipulation of the knowledge base based on association patterns mentioned above. To serve this purpose, K provides the context looping statement in the form `"context <pattern> [where <exp>] [select <vars>] do <statements> end_context,"`

which iterates over each extensional pattern of the sub-knowledge-base (context) specified by <pattern>. The optional where-clause is used to specify a boolean expression as the inter-class selection condition so that the iteration will skip those tuples that do not satisfy the condition. One can also use the optional select-clause to specify a projection over only those columns (represented by their range variables) that he/she is interested and thus eliminate the resulting redundant tuples. When no select-clause is given, implicit projection will be performed over those columns, each of which defines a range variable. For example, the following statement will print the name of each professor whose age is smaller than that of any his/her advised student. Note that the where-clause is to used to specify the inter-class selection condition between Grad and Professor, and the select-clause is used to project over Professor column and remove the redundant tuples so that each qualified professor will appear only once even if he/she advises more than one student.

```
context g:Grad *>[advisor] p:Professor where g.age > p.age
  select p
    do p.name.display();
end_context;
```

Note that by introducing the notation of range variables, no alias class [OQL89b] is necessary in an association pattern. For example, "a1:A *>[Pa] B *>[Pb] a2:A" defines two range variables a1 and a2 over the same class A. As context looping

statement is just one of the control constructs provided by K, it can be freely combined with other constructs or nested in an arbitrary number of levels to comply with the principle of orthogonality. This is one important issue that must be addressed in the design of a knowledge-base programming language but not necessarily in that of a query language. For example, the following statement prints the name of the chairman of each department followed by all the other faculty members of that department. The first context statement identifies the Department class, then the nested context statement is applied to each department. For each looping of the nested context statement, we identify all the professors who are faculty members of the particular department, and print his/her name if he/she is not the chairman.

```
context d:Department
  do d.chairman.name.display();
    context p:Professor *>[faculty_of] d
      do if p != d.chairman
        then p.name.display();
      end if;
    end context;
  end context;
```

4.3.2 Navigation in Object-oriented Knowledge Base

In existing object-oriented data-base systems, navigation is expressed by using the dot expression for implicit joins. However, the use of dot expression is limited by the following factors. First, navigation is done only in one direction unless inverse attributes are supported and explicitly defined

in the system. Second, navigation cannot continue when a multi-valued attribute is met. For example, the dot expression "s1.friends.enroll" is not allowed if "s1" is a Student instance, and both "friends" and "enroll" are set-valued aggregation associations defined from Student to Student, and from Student to Course, respectively. The reason is that the expression "s1.friends" returns a set of Student instances, and dot expression only allows "enroll" to be applied to a single Student instance. Third, dot expression cannot express navigation via negation, or the "non-association" relationships. For example, it is not possible to express "all the courses that student s1 does not enroll" using simple dot expression. Last, it is impossible (except using dynamic binding) to explicitly navigate via the generalization association in order to access different representations of the same object in different classes. K supports all the above cases using the association patterns described in Section 4.2.

Bi-directional navigation. As mentioned in Section 4.1, any aggregation association between two entity classes are interpreted as a bi-directional link, and any generalization association can also be thought of as a bi-directional link from a superclass to a subclass (generalization) or vice versa (specialization). Thus, one can navigate from one class to another by following these bi-directional links to form a linear association pattern of arbitrary length. For example, one can use the following context looping statement to print

all the course titles that the advisor of "s1" is teaching. Note that "instructor" is an aggregation association defined by "Course" and therefore we use "*<[instructor]" to express the navigation from Professor to Course.

```
context s1 *>[advisor] Professor *<[instructor] c:Course
  do c.title.display();
end_context;
```

Multi-valued navigation. Both single-valued and multi-valued aggregation associations are expressed uniformly in an association pattern. For example, one can use the following context looping statement to print the names of all the courses that student s1's friends enroll. Note that here, we use implicit projection to remove the redundant courses that are taken by more than one of s1's friends.

```
context s1 *>[friends] Student *>[enroll] c:Course
  do c.name.display();
end_context;
```

Navigation via negation. By using the non-associate ("!") operator, one can traverse via the conceptual "non-associate" links through the knowledge base. For example, the following program will print all the course titles that "s1" does not enroll and come out with a set C1 containing all the courses that "s1" does not enroll for further manipulation. Note that the association pattern returns (i) student "s1" if he/she does not enroll in any course, and (ii) all the courses that "s1" does not enroll. The normalized relation will contain one

of the following types of tuples: (i) $\langle s1, \text{Course} \rangle$, if "s1" does not enroll in any course, (ii) $\langle \text{null}, \text{Course} \rangle$, if "s1" enrolls in some, but not all of the courses, and (iii) $\langle \text{null}, \text{null} \rangle$, if "s1" enrolls in all the courses. We then use the range variable "c" to loop over all (if any) the selected Course instances.

```
local C1: set of Course;
context s1 !>[enroll] c:Course
  do c.title.display();
    C1 := C1 + c; /* add "c" into set "C1" */
end_context;
```

Generalization navigation. In order to support static (instead of dynamic) type checking, K allows the user to explicitly express navigation via the generalization association to access different representations of the same object in different classes. For example, the following statement will print the name of each student and, if this student is also a graduate student, print the name of his/her advisor (if any). Note that the normalized relation returned by the context expression could contain three types of tuples: (i) $\langle \text{Student}, \text{null}, \text{null} \rangle$, if a particular student is not a graduate student, (ii) $\langle \text{Student}, \text{Grad}, \text{null} \rangle$, if a particular student is a graduate student but does not have an advisor, and (iii) $\langle \text{Student}, \text{Grad}, \text{Professor} \rangle$, if a particular student is a graduate student and has an advisor. It is the user's responsibility to check the case of null value to avoid a run time error when trying to send a message to null as in C++.

Also note that similar to CLU [LIS77], we use different range variables to represent different instances of the same object in different classes so that type checking can always be done at compilation time.

```
context s:Student *>[G] g:Grad *>[advisor] p:Professor
  do s.name.display();
    if (g != null) and (p != null)
      then p.name.display();
    end if;
end context;
```

Similarly, one can use the cast operator "\$" mentioned in Chapter 3 to implement late binding. For example, suppose "Employee" has two subclasses "TA" and "Professor," both of which define a method called "raise." Then, one can use the following context looping statement to give a raise to every employee. Note that in the case statement, we use the cast operator to test if an employee is either a TA or a Professor, and apply the appropriate "raise" method to the corresponding TA or Professor instance. In the later version of K that supports virtual class as in C++ [STR86], the testing and casting will be performed automatically by the system.

```
context e:Employee
  do case
    when TA$e != null do TA$e.raise();
    when Professor$e != null do Professor$e.raise();
  end case;
end context;
```

4.4 Existential and Universal Quantifiers

Statements for the retrieval and manipulation of a knowledge base may involve existential and universal quantifiers in the form "exist <vars> in <context> [suchthat <boolean>]" and "forall <vars> in <context> suchthat <boolean>." Quantifiers based on association patterns make it much easier for the users to declaratively pose logic questions upon the knowledge base. An existential quantifier returns true if there exists any object instance(s) denoted by <vars> in the sub-knowledge-base specified by <context> and satisfies the optional <boolean>. For example, the following expression tests if there exists any student who is taught by his/her own advisor: "exist s in s:Student and (*>[enroll] Course *>[instructor] p1:Professor, *>[advisor] p2:Professor) suchthat p1 = p2." Note that the "suchthat" clause of an existential quantifier can be omitted if it can be specified as an intra-class selection condition in the association pattern. For example, one can ask if a student whose name is "John Smith" takes any CIS course as "(exist c in Student[name = \"John Smith\"] *>[enroll] c:Course[offered_by.name = \"CIS\"])." It is also possible to specify more than one variables in a quantifier. For example, "exist s,p in s:Student !>[advisor] p:Professor" returns true if there exists both any student who does not have an advisor and any professor who is not advising any student. Note that the

result of "exist a in a:A !>[p] B" and "exist b in A !>[p] b:B" is not necessarily the same. Similarly, a universal quantifier returns true if all the object instances denoted by <vars> in the sub-knowledge-base specified by <context> satisfy <boolean>. For example, one can ask if all the students have GPA greater than 3.5 as "(forall s in s:Student suchthat s.GPA > 3.5)." As a special case, if no object instance exists in the sub-knowledge-base, then a universal quantifier returns false by default. Note that in simple cases, quantifiers are not always necessary. For example, to test if student "s1" has any advisor, one can use either "s1.advisor != null" or "exist p in s1 *>[advisor] p:Professor." However, as pointed out in Section 4.3, the use of dot expression has many limitations and cannot express many complex cases concisely. For example, the following expression tests if student "s1" and the friends of "s1" enroll in any common course: "exist c in s1 *>[friends] Student *>[enroll] c:Course *< [enroll] s1." This question is difficult to express using traditional dot expression and set operations. Also note that we do not support the "in" operator to test the membership of an object instance within a set in the first version of K. The reason is that whatever the "in" operator can express can always be expressed by using quantifiers, but the vice versa is not always true due to the limitations of dot expression described above. For example, to ask "if x

enrolls in course y," one can use the expression "exist c in x *>[enroll] c:Course suchthat c = y."

As quantified expressions themselves are boolean expressions too, they can be nested in an arbitrary number of levels or combined with other constructs wherever boolean expressions are allowed. For example, the following statement will print the names of all the students who take all the courses currently offered by his/her major department.

```
context s:Student *>[major] d:Department
  do if forall c1 in c1:Course *>[offered_by] d
    suchthat exist c2 in s *>[enroll] c2:Course
      suchthat (c1 = c2)
    then s.name.display();
  end_if;
end_context;
```

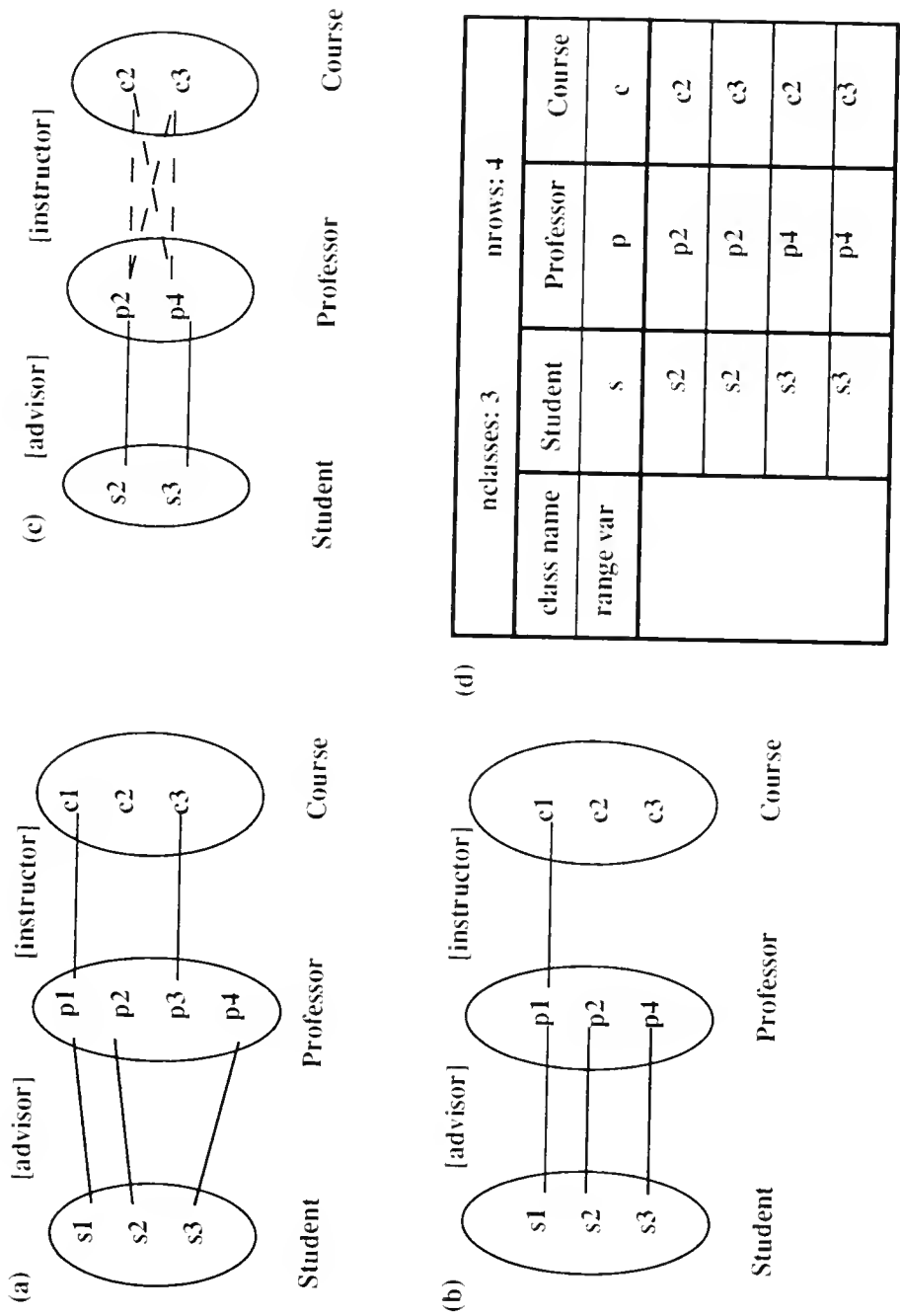


Figure 4.1 The Evaluation of the Association Pattern
s:Student *>[advisor] p:Professor !<[instructor] c:Course

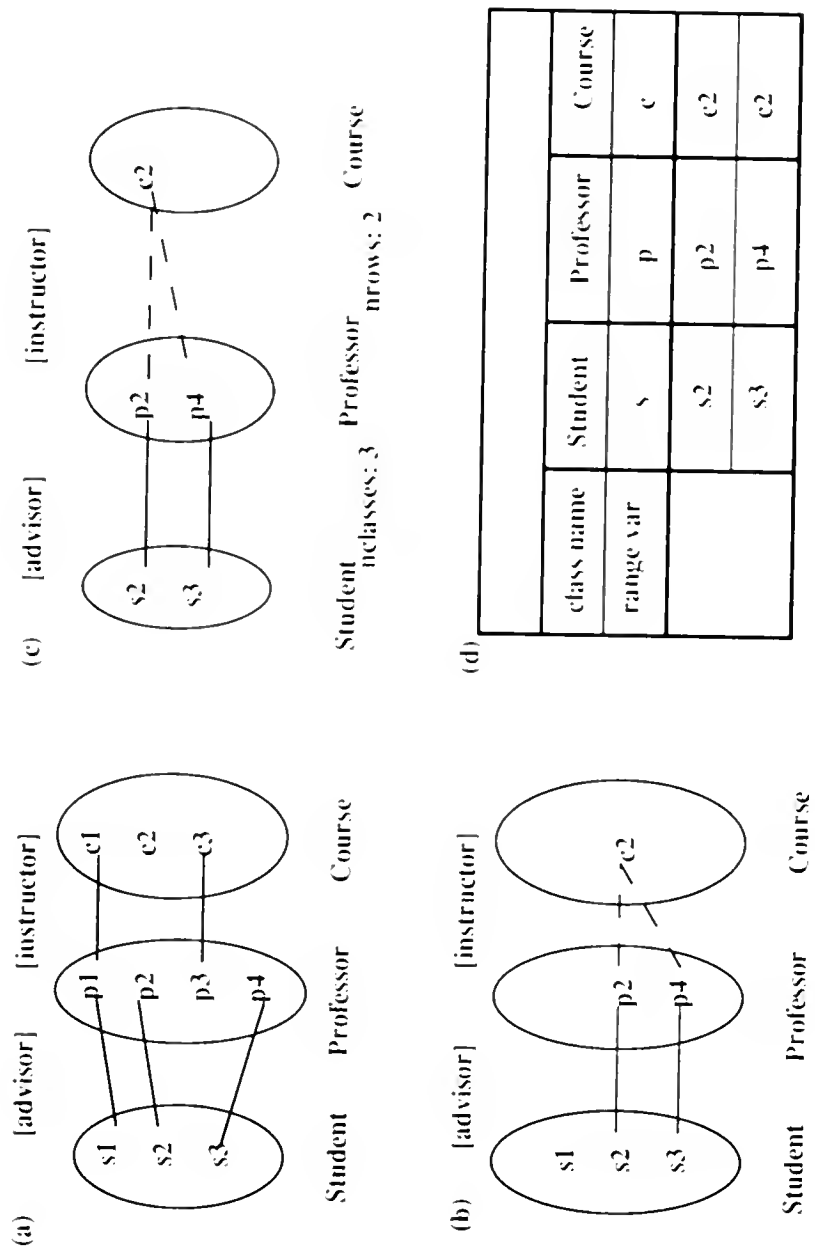


Figure 4.2 The Evaluation of the Association Pattern
s:Student *>[advisor] (p:Professor !<[instructor] c:Course)

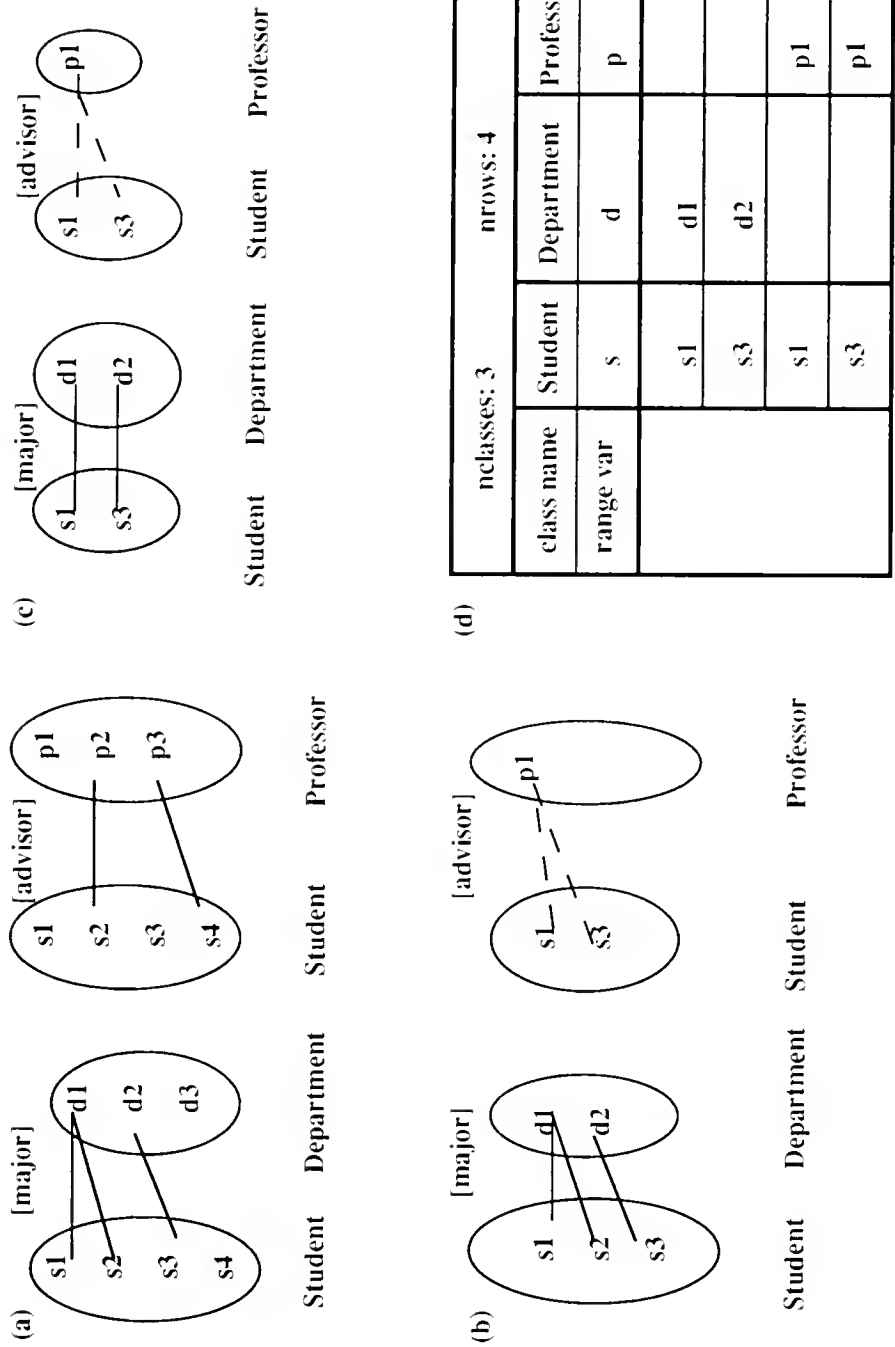


Figure 4.3 The Evaluation of the Association Pattern
s:Student and (*>[major] d:Department, !<[advisor] p:Professor

CHAPTER 5

BEHAVIORAL ABSTRACTION MECHANISMS

As mentioned in Chapter 3, behavioral properties of objects are modeled by methods and rules. Corresponding to each executable software system, the user also has to define a named K program (similar to the "main" program of C++) as the starting point of execution. Each program is defined in the form of "program <identifier> is <statement> {<statement>} end [<identifier>;]." Note that such programs are defined in parallel with object classes. A detailed description of methods and rules will be given in this Chapter.

5.1 Method Definition

At the language layer, each method is specified in the method section and implemented in the implementation section of its defining class in the following syntax:

```
methods:
  public: | private: | protected:
    <method_specification>; {<method_specification>;}
  {public: | private: | protected:
    <method_specification>; {<method_specification>;}
  }

implementations:
  <method_implementation>; {<method_implementation>;}
```

Each `<method_specification>` is the signature of a method in the form "method | operator (parameter> {,<parameter>}) : [<collection> of] <return_type>." Note that <return_type> could be a class name or "void" (which is used to specify that there is no return value of this method). Each parameter is specified in the same way as a variable declaration as "<var> : [<collection> of] <class>," where <collection> is either set, list, or array[<integer>], and <class> is the type of the parameter. Each `<method_implementation>` is defined as "`<method_specification>` is <statements> end [<method_name>];," where <statements> represents a sequence of K computation statements in the form "<statement>; {<statement>;}." One can also overload any system-defined relational operator (>, <, =, !=, >=, <=), arithmetic operator (+, -, *, /, mod), set operator (+, -, &), logic operator (and, or), or unary operator (+, -, not) by redefining the operator in a user-defined entity class or domain class. Operator specifications and implementations are syntactically the same as those of other methods, except the key word method is replaced by operator. The behavior of an operator therefore depends on the object to which it is applied (operator polymorphism). The difference between methods and operators is that while methods are invoked generally by using the dot expression, operators are invoked by using the traditional binary and unary expressions. A detailed description of various types of expressions will be given in the following of this Section.

The body of a method is a sequence of K computation statements, which can be categorized as follows:

(1) Expression. The most simple form of K statements is "<expression>," which can be further categorized as follows:

(1.1) Single item. A single item could be (i) an identifier (e.g., a local variable or attribute name in an intra-class selection condition), (ii) a primitive domain class object, i.e., integer (e.g., 12), real (e.g., 2.5), boolean (e.g., true), character (e.g., 'A'), and string (e.g., "Smith"), (iii) a complex domain class object (e.g., date(mm := 12, dd := 31, yy := 1990)), or (iv) a method invocation when the receiver of the method is omitted (when the receiver is "this" in a method body or in an intra-class selection condition). The syntax for a method invocation is "<identifier>([<expression>{,<expression>}])," where each <expression> corresponds to a formal parameter defined by the method. Note that an identifier starts with a letter or special symbol ('_', '#'), followed by any number of letters, digits, or special symbols.

(1.2) Dot expression. Traditional dot expression is supported in K for simple navigation in one of the following two forms: (i) "<expression>.<identifier>," which is used to access an attribute referred by <identifier> of the object instance returned by <expression>, or (ii) "<expression>.<method>," where <method> is a method invocation applied to the object instance returned by <expression>. For

example, "x.advisor.name" returns the name of the advisor of a student referred by "x." Similarly, "x.eval_GPA()" returns the GPA value of the student referred by "x," where "eval_GPA" is a method defined by class Student and takes no parameter.

(1.3) Assignment expression. One can use the assignment expression to update an attribute of an object instance in the form "<expression>.<identifier> := <expression>," or to assign a value to a local variable in the form "<identifier> := <expression>". For example, the expression "x.age := 20" updates the "age" attribute of a student instance referred by "x" to be 20.

(1.4) Binary expression. Binary operators, which include relational operator (>, <, =, !=, >=, <=), arithmetic operator (+, -, *, /, mod), set operator (+, -, &), and logic operator (and, or), can be used to form binary expressions in the form "<expression> <op> <expression>." As mentioned in Section 3.4, cast operator '\$' can be used to ascribe a type (i.e., a class name referred by an identifier) to an expression in the binary form "<identifier>\$<expression>."

(1.5) Unary expression. Unary operators '+', '-', and "not" can be used preceding an expression to form an unary expression in the form <op><expression>. Note that unary operators have higher precedence over binary operators.

(1.6) Array expression. One can use the array expression "<expression>['<expression>']" to access a particular element indexed by the second <expression> from an array

returned by the first <expression>. Multi-dimensional arrays will be supported in the later version of K.

(1.7) Object expression. As mentioned in Section 3.5, one can use the "new" and "pnew" operator to create transient and persistent object instances of certain entity class referred by an identifier and return the new iids in the form "new | pnew <identifier> ([<assignment> {,<assignment>}])." Note that the optional assignment expressions are used to assign values to some of the visible attributes, depending on where this new object is created as described in Section 3.1. As shown in Section 3.5, one can also use the "insert" operator to create a new instance in a certain class for an existing object instance referred by an expression and return the new iid in the form "<expression> insert <identifier> ([<assignment> {,<assignment>}])".

(1.8) Quantifier expression. As mentioned in Section 4.4, existential and universal quantifier expressions are boolean-valued expressions in the form "exist <identifiers> in <pattern> [suchthat <expression>]" and "forall <identifiers> in <pattern> suchthat <expression>," where <identifiers> is a list of variables in the form "<identifier>{,<identifier>}".

(1.9) Parenthesis expression. Any expression can be enclosed in a pair of parentheses as (<expression>) so that it can be evaluated as a single expression without being effected by its neighboring operators that have higher

precedence. We show the precedence priority of all the operators in ascending order as follows:

```

level 1:  :=
level 2:  new, pnew
level 3:  or
level 4:  and
level 5:  >, >=, <, <=, =, !=
level 6:  +, - (binary plus/union, minus/difference),
          & (intersection)
level 7:  *, \, mod
level 8:  not, +, - (unary plus and minus)
level 9:  insert
level 10: . (dot operator)
level 11: $ (cast operator)

```

(2) Block statement. One can define a sequence of statements as a block in the form "[local <vars>] begin <statement> end." Similar to Ada [DOD83], local variables can be declared with a block following the key word "local" in the form "<var>; {<var>;}," where each <var> is a variable declaration in the form <identifier>{, <identifier>}: [<collection> of] <class>." Note that the scope of a local variable is limited to the end of the block in which the variable is declared.

(3) Conditional statement. Corresponding to the Testing (T) control association, two types of conditional statement are provided: (i) if-then-else statement in the form "if <expression> then <statements> [else <statements>] end_if," and (ii) case statement in the form "case when <expression> do <statements> {when <expression> do <statements>} [otherwise do <statements>] end_case." Note that as conditional statements

can be nested, the use of the key words "end_if" and "end_case" avoid the "dangling else" problem [AH086] by defining a clear scope for each statement. Also note that any expression that returns a boolean value (including the quantifier expressions described in Section 4.4) can be used as the test condition.

(4) Repetitive statement. In addition to the context looping statement "context <pattern> [where <expression>] [select <vars>] do <statements> end_context" described in Section 4.3, two types of repetitive statement are provided for iteration: (i) for statement in the form "for <expression> until <expression> [by <expression>] do <statements> end_for," and (ii) while statement in the form "while <expression> do <statements> end_while." The first <expression> following the key word "for" will be an assignment expression that initializes an iteration variable. The for-statement iterates over <statements> until the value of the iteration variable satisfies the <expression> following the key word "until." After each iteration, the iteration variable is updated by executing the assignment expression specified after the key word "by." The default setting is "increased by 1." For example, "for i := 1 until (i > 10) by i := i*2 do i.display() end_if;" will print 1, 2, 4, and 8. Note that the type of the iteration variable can be any domain class that has the following operators defined: plus (+) for increment, minus (-) for decrement, assignment (:=) for initialization, and equal

(=) for comparison. The while statement will iterate over <statements> as long as the expression that follows the key word "while" returns "true."

(5) Flow statement. Inside a repetitive statement, one can use the "break" statement or "continue" statement to alter the control flow. The break statement causes the control to exit the repetitive structure, while the continue statement ignores the statements following the continue statement and forces the control to go to the beginning of the repetitive structure. One can also use the "return [<expression>]" statement to terminate the execution of a method normally and return the control flow to the point where the method is invoked.

(6) Object statement. As mentioned in Section 4.5, one can use the "delete <expression>" and "destroy <expression>" statements to manipulate the knowledge base. The delete statement deletes an entity class instance returned by <expression> as well as all the instances that have the same oid from all the subclasses of the entity class. For each instance being deleted, any association with other entity class instance is also deleted. The destroy statement, on the other hand, recursively deletes not only the entity class instance returned by <expression>, but also all the instances that have the same oid as <expression> from all the superclasses and subclasses of the entity class.

(7) Rule statement. Within a user program, one can use the "deactivate <identifier>" and "activate <identifier>" statements to deactivate/activate a rule whose rule name is referred by <identifier>. A detailed description of rules will be given in Section 5.2.

(8) Abort statement. As objects are manipulated by methods in object-oriented paradigm [STE86], we feel that it is a natural way to consider the execution of each method as a single transaction, which commits when the method terminates. We also provide the user with the "abort" statement to undo any update to the knowledge base (i.e., the states of persistent entity class instances) made before the abort statement, and after either (i) the beginning of the method execution (if no abort statement has been executed) or (ii) the previous abort statement executed within the method execution. Note that the abort statement has no effect on the update to the value of any local variable itself or to the state of any transient entity class instance. Also note that the abort statement does not alter the normal control flow. The execution of a method continues as a new transaction and all the updates made after the last abort statement will be committed to the knowledge base when the method terminates. A more detailed description of the execution model will be given in Chapter 6.

5.2 Rule Definition

Rules serve as a high-level mechanism for specifying declarative knowledge that governs the manipulations of objects made by KBMS operations, updates, and user-defined methods. Rules are specified in the following syntax:

```

rule <identifier> is
    triggered <trigger_conditions>
    [condition <guard_expression>]
    [action <statements>]
    [otherwise <statements>]
end [<identifier>]

```

Each rule is given a name for its identification, which must be unique within its defining class. Each rule is specified by a set of trigger conditions and a rule body. Trigger conditions are specified as "<trigger_condition> {; <trigger_condition>}," where each <trigger_condition> consists of a timing specification and a sequence of knowledge-base event specification in the form "<timing> <event> {, <event>}." Timing specification (or coupling mode) can be "before," "after," or "immediate_after." Event specification can be a (1) KBMS operation (new, pnew, insert, delete, and destroy), (2) update ("update [<class>::] <identifier>"), where <identifier> is an attribute defined or inherited by the defining class of the rule, or (3) user-defined method ("[<class>::] <identifier> ()"), where <identifier> is a method defined or inherited by the defining class of the rule. The two-colon operator ("::") is used to specify from which

superclass of the defining class that a particular attribute or method is inherited when name conflict occurs. Note that the two-colon operator is used only in the trigger conditions of rules, not in the computation statements where the cast operator ('\$') should be used for conflict resolution as described in Section 3.4. Internally, the system uses the event specification as the key of a in-memory rule hash table for fast retrieval of applicable rules as will be described in Chapter 6.

Note that certain combinations of timing and event are semantically invalid. For example, "before new/pnew/insert" and "after delete/destroy" are meaningless because each rule is applied to some instance of its defining class and there is no such instance exists for the above trigger conditions. For domain classes, "pnew/insert/delete/destroy" are not valid events as they are not applicable to values. However, we allow the use of "new/update" in a trigger condition to specify constraints that govern the values in a range or enumeration.

It is important that the use of "new/pnew," "insert" and "update <identifier>" not be confused. In general, rules govern the values of certain attribute should be specified by using "update <identifier>" as the event instead of using "new/pnew" or "insert." The reason is that any assignment of attribute value is treated as an update operation, even if it is embedded in a new, pnew, or insert operation. Similarly, for those rules that ensure that certain attributes can not

be null (i.e., non-optional) when an instance is created, one should use "insert" instead of "new/pnew" for entity classes, and use "new" for domain classes. The reason is that creating a new entity class object of class "X" implies the insertion of corresponding instances to class "X" as well as all the superclasses of "X," while inserting an instance does not imply creating a new object as described in Section 3.5.

We do not support "retrieve" as a valid KBMS operation specification for the following reasons. First, as far as the knowledge-base consistence is concerned, the retrieve (read) operation will not change the state of the knowledge base. Second, as we expect there are many retrieve operations in a complex K program, the performance will be significantly reduced if the system has to check rules for every retrieve operation. Third, in the case that the user would like to trigger some action when retrieving certain attribute, he/she can apply the object-oriented encapsulation principle by defining this attribute as a "private" attribute and defining a public/protected method as the interface to retrieve this attribute. Rules associated with this method can then be defined to perform the triggering.

The rule body consists of (i) "condition" clause that is a guard expression, and (ii) "action," and "otherwise" clauses, both of which can be a sequence of any K computation statement described in Section 5.1. Each guard expression is in the form "[<guard> {, <guard>} |] <target>" and the

evaluation of a guard expression can return either (i) true: if all the guards and the target (all of which are boolean expressions by themselves) are true, (ii) skip: if any of the guards is false when they are evaluated from left to right, (iii) false: if all the guards are true but the target is false. Note that the guard expression itself is not boolean expression, and it can be used only in the condition clause of rule definitions instead of computation statements. Also note that, although the semantics of a guard expression can be implemented by nesting of if-then-else constructs, the guard expression is a simpler and more concise construct to use, particularly when the number of guards is large. Besides, we feel that rules should be specified as declaratively as possible, and we would like to make a clear distinction among the "condition," "action," and "otherwise" parts of a rule instead of mixing them in a nested if-then-else procedural statement. This clear distinction also makes it possible for different implementation strategies. For example, the "condition" and "action" (or "otherwise") parts can be executed as separate transactions as in HiPAC [HSU88, CHA89] and ODE [GEH91]. Similar to method invocation, rule checking is performed at the instance level, and the pseudo variable "this" can be used in a rule body to represent a certain instance of the defining class to which some event occurs as shown in Figure 3.1.

We modified the syntax of the OQL rule language [ALA90] so that it can be seamlessly incorporated into K. The difference between the K rule language and the OQL rule language is three-fold. First, we use a uniform syntax that is similar to the ECA (event-condition-action) rule of Chakravarthy [CHA89] to represent both state rules (constraints) and operational rules (triggers) for simplicity. Second, we use the guard expression to subsume the semantics of "if <pattern1> exists, then <pattern2> must also exist, otherwise do some corrective_action" in the OQL rule language so that we can provide more expressive power (e.g., any number of guards can be specified, and any boolean expression including quantifier can be used as a guard or target) and avoid the confusion with the "if-then-else" computation statement. Third, we subsume the implicit set-oriented semantics in the OQL rule language "if context <pattern> then <operation>" and "if context <pattern> then <pattern> corrective_action <operation>," where the use of the key word "context" implies that this rule will be checked against all the instances of the defining class of the rule. To specify the same rule in K, quantifier expression can be used to test the existence of certain patterns, and context looping statement can be used in the action- or otherwise- clause to iterate over certain context. Note that the OQL rule language is more declarative in the sense that the user does not have to deal with the detailed specification of control structure;

however, this feature is undesirable in an integrated knowledge-base programming language because it overloads the semantics of the "if-then-else" computation statement and prohibit the user to have finer-grained control over the computation at the instance level in terms of using various control structures and variable bindings. Besides, when more than one pattern is specified (e.g., in a guard expression), the semantics of looping over several contexts at the same time is undefined. To sum up, K provides consistent syntax, more expressiveness, and finer-grained control of the rule actions over OQL rules.

All the rules are assumed to be active when a user session begins. However, during the execution of a user program, one can use the "activate <identifier>" or "deactivate <identifier>" statements to temporarily activate or deactivate any particular rule referred by <identifier>, respectively. Note that rules are treated as protected properties and are encapsulated in their defining classes. Therefore, each rule of class "X" can be referred by the activate/deactivate statements only in those method bodies and rule bodies of class "X" or any subclass of "X." For each knowledge-base event occurs to instance "this" of class "X," all the applicable rules will be triggered (i.e., the evaluation of the rule body) according to the trigger conditions of each rule at either (i) before the triggering event, (ii) immediately after the triggering event, or (iii)

not immediately after the triggering event, but at the end of the parent event that causes the triggering event. A detailed description of how to find applicable rules will be given in Chapter 6. Note that the use of "after" mode allows for temporary violation of constraints (which is likely to happen when a constraint on an object depends on two inter-related values and when one of the values is updated) by deferring the rule checking until the end of a higher level operation. In the case that multiple rules satisfy a trigger condition, such rules will be triggered in some unspecified order, which is dependent on the implementation. Later version of K will allow the user to explicitly specify the triggering orders in terms of rule associations, i.e., sequential, parallel, and synchronization relationships among rules.

The rule body of each rule is evaluated as follows: (i) if the condition-clause returns true, then the action-clause (if provided) is executed, (ii) if the condition-clause returns skip, then do-nothing, and (iii) if the condition-clause returns false, then the otherwise-clause (if provided) is executed. For example, the rule CIS_rule1 specified in Figure 3.1 will be executed at the end of those methods that are applied to a student instance and update the major of this particular student. The otherwise-clause will be executed if this particular student is a CIS major (guard is true) and his/her GPA is not greater than 3.0 (target condition is false). Similarly, General_rule1 will be checked after the

method "suspend." A detailed description of the computation model will be given in Chapter 6.

As a final note, rules specified in a class definition can conflict with other rules for the same knowledge-base event or cause infinite looping during execution. In general, it is not possible to automate the validation of rules in the context of knowledge-base programming language. It is the user's responsibility to make sure such logic errors do not happen as in any programming language.

CHAPTER 6 COMPUTATION MODEL

A computation model, akin to a virtual machine, provides the operational semantics of a programming language. To meet the various requirements of different application domains, a multi-paradigm computation model is needed for supporting object-oriented, parallel, non-determinism, and rule-based computations and, at the same time, satisfying the concurrency control and recovery requirements of the knowledge-base management system. In this chapter, we present the computation model of K, which satisfies the above requirements in an integrated fashion.

6.1 Overview

The computation model of K is based on object-oriented paradigm [STE86, ACM90] and nested transaction [MOS81, HSU88] to model the behavior of the combined execution of methods and triggered rules in an object-oriented framework.

The nested transaction model provides a mechanism for coping with failures and introducing concurrency in distributed applications [MOS81]. It also provides an ideal framework for modeling cooperate users and long-lived activities in software development environment [DAY90]. The

execution of each method is considered as a transaction, i.e., an atomic action against the knowledge base: once invoked, it either complete all its operations or behaves as if it were never invoked. Transactions can be nested to an arbitrary number of levels by defining a new method, which in turn invoke other pre-defined methods. As a result, a transaction may contain any number of nested transactions or sub-transactions, some of which may be required to perform sequentially, some concurrently, and all are organized as a transaction tree whose root is the top-level transaction. Changes to the knowledge base made by a nested transaction are contingent upon the successful commitment of all of its ancestral transactions. Aborting any of its ancestors invalidates all of its changes. If a nested transaction aborts, the knowledge-base state seen by its parent is the same as it was immediately prior to starting the nested transaction. Note that the abort of a transaction has no effect on any update made to the value of a local variable itself or to the state of any transient entity class instance. Concurrency control and recovery will be supported by the storage layer of the KBMS using the traditional pessimistic transaction mechanism [ONT91].

The triggering of a rule (i.e., the execution of a rule body) is also considered as a sub-transaction of the triggering transaction (i.e., the transaction that represents the execution of a method that triggers the rules) or the

parent of the triggering transaction, depending on the coupling modes (before, after, immediate_after, and in_parallel). Nested firing of rules is supported by the rule handler of the KBMS, which systematically handles rules triggered within the execution of another rule. A detailed description of the computation paradigm will be given in Section 6.2.

Note that though parallelism is not supported in the current version of K, it can be easily incorporated into the computation model as follows. First, inter-object parallelism (each active object can execute independently of other objects) and intra-object parallelism (an active object can host several threads of execution concurrently) [ELL89] can be modeled by treating (i) active objects as independent transactions all of which are executed in parallel, and (ii) multiple threads of execution as concurrent sub-transactions, respectively. Second, asynchronous message passing can be supported by allowing the parent and children transactions to be running concurrently as long as they do not read or write the same data item. Third, operations can be applied in parallel both implicitly (e.g., concurrent execution of multiple users' programs) and explicitly (e.g., specified by some parallelism constructs such as "parbegin...parend"). Fourth, multiple rules with disjoint domains can be triggered in parallel, and multiple rules triggered by the same event can be executed concurrently.

Scheduling non-determinism (non-determinism in time) can also be incorporated into the computation model naturally. If time limit or priority is not constrained in multiple methods that are waiting to be executed, then these methods will be executed as concurrent sub-transactions under the constraint of serilizability, i.e., the net result is equivalent to some serial order execution. The system can verify the correctness of the non-deterministic execution if there is any pre-defined rule that specifies the post-condition of a particular method. Secondly, when time limit or priority is not constrained in multiple production rules that are triggered simultaneously and with disjoint domains, then their actions will be executed in the same way as above.

6.2 Extended Object-oriented Computation

Traditional object-oriented paradigm supports message passing with built-in inheritance mechanism. We proposed an extended paradigm to incorporate the execution of rules and any structural association types in an object-oriented knowledge-base management system that supports the knowledge model described in Chapter 3. The occurrence of a knowledge-base event "P" on instance "this" of class "C" consists of the following steps: (1) get, bind, and trigger all the applicable "before" rules of "P," (2) execute the event "P" itself, (3) get, bind, and trigger all the applicable "immediate_after" rules of "P," and (4) get, bind, and trigger

all the applicable "after" rules of those next-level events whose occurrences are nested in event "P." Note that the execution of a rule body might invoke certain events that might in turn trigger other rules recursively. All the activities from step 1 to step 4 are organized as a transaction tree as shown in Figure 6.1.

As mentioned in Chapter 5, a knowledge-base event "P" could be a KBMS operation (new, pnew, insert, delete, and destroy), an update, or a method invocation. After compilation, each triggering event is represented as either <id, SourceClass, operation> for both KBMS operations and method invocations, or <id, SourceClass, update, operand> for updates, where "id" is the instance to which the event occurs, and "SourceClass" could be (i) the class one of whose instances is being created, deleted, destroyed, or inserted, (ii) the defining class of the method that is being invoked, or (iii) the defining class of the attribute that is being updated. Note that in the case of inherited attributes or methods, "SourceClass" will be a superclass of the class to which "id" belongs. Similarly, the event specifications of rules are represented as <SourceClass, operation> or <SourceClass, update, operand>. In general, applicable rules of event "P" with certain coupling mode must satisfy the following conditions. First, the coupling mode and knowledge-base event must match one of the trigger conditions of this rule. Second, there must be an instance with the same oid as

"id" in the defining class of this rule. In other words, for any triggering event $\langle \text{id}, \text{SourceClass}, \text{operation} \rangle$ or $\langle \text{id}, \text{SourceClass}, \text{update}, \text{operand} \rangle$, we match the triggering event with all the active rules of "SourceClass" and any of its subclasses that has an instance with the same oid as "id."

The advantage of this approach is three-fold. First, the search space is reduced in the sense that we start from "SourceClass" instead of the root class to avoid searching those inherited rules that are impossible to match the triggering event. For example, the event "update Student::GPA" will never trigger any rule defined by class "Person" because "GPA" is defined by "Student" and not visible from "Person." Second, re-defined attributes or methods can be easily identified and thus avoid incorrectly triggering rules. For example, if class "TA" re-defines the method "evaluate" of "Grad," then applying "evaluate" to a TA instance will not trigger any rule of "Grad", which has "Grad::evaluate()" as its trigger condition because the SourceClasses are different. Third, different from existing rule-based systems such as ODE [GEH91] and HiPAC [CHA89], the choice of applicable rules in our paradigm is not limited to inherited rules because any entity class object could have multiple representations in multiple classes as described in Chapter 3. For example, suppose the GPA of a student must be greater than 2.0, the GPA of a TA must be greater than 3.0, and both rules have "update Student::GPA" as one trigger condition. Then, the update of

the GPA value of a Student instance must trigger both rules of Student and TA as long as this student is also a TA. Failing to trigger the TA rule may leave the knowledge base in an inconsistent state. None of the existing systems addresses this problem adequately.

In the case that an update is made to a complex domain class object that is embedded as part of a value attribute of some entity class instance, such an update will be rippled back to all the enclosing domain classes and finally the hosting entity class via the navigation path. For example, suppose class "Person" defines a value attribute called "birthday" whose type is a complex domain class "date." Class "date" defines three value attributes "month," "day," and "year," all of which have type "integer." Then, the event "p.birthday.year := 1972" should trigger not only the "update date::year" rule (if any) defined by "date," but also the "update Person::birthday" rule (if any) defined by "Person."

Note that the behavior of KBMS operations with respect to various association types is either built into the object manager (for generalization and aggregation of the kernel model) or governed by generic rules (for user-defined association types as will be described later). For example, the triggering event of a "before delete" rule of Person is represented as <Person, delete>, which is different from that of a "before delete" rule of Student. Thus, any "before delete" rule of "Person" will not get triggered when a Student

instance is deleted because deleting a Student instance will not delete the corresponding Person instance as described in Section 3.5. On the other hand, when a Student instance is getting deleted, the object manager will automatically delete the corresponding TA instance (if any) and thus trigger any "before delete" rule of TA.

Note that if an "abort" statement is executed in the action- or otherwise-clause of a rule, then its effect depends on the timing when the rule is triggered as follows:

(1) If the rule is triggered "before" the triggering event "T," then "T" will not be performed and directly return null.

(2) If the rule is triggered "immediate_after" the triggering event "T," then "T" itself will be aborted. The abort of "T" will have the following effects: (i) all the rules that have been triggered by "T" will be aborted, (ii) all the rules waiting to be triggered at the end of "T" will be discarded, (iii) all the "after T" rules waiting to be triggered at the end of the parent event of "T" will be discarded, and (iv) all the events that are nested within "T" will also be aborted recursively.

(3) If the rule is triggered at the end of the parent event of "T," say, "T*," then "T*" itself will be aborted.

Also note that though not supported in the current version of K, generic rules can be incorporated into the computation model easily to facilitate model extensibility.

All we have to do is to extend the choice of applicable rules for a KBMS operation $\langle \text{SourceClass}, \text{operation} \rangle$ to include bound generic rules from those association objects in which "SourceClass" participates as either the defining class or one of the constituent classes. Generic rules can be bound at either compile time or execution time, depending on the implementation. For example, one can define a new association type "Interaction" [SU89b] by specifying a generic rule for referential constraint, which says that before deleting an instance from class $\langle \text{self} \rangle$, which is one of the constituent classes of an Interaction association, we must delete from the defining class of this Interaction association all the instances that are associated with this instance by the Interaction association. Suppose there is an Interaction association from class "Advising" to "Grad" and "Professor," which models the n:1 relationships between Grad and Professor. Then, before deleting an instance of "Grad," which is denoted by "this," the system will query the dictionary and find that "Grad" is the constituent class of an Interaction association defined by "Advising." The system will then bind the above generic rule of class "Interaction" by replacing $\langle \text{self} \rangle$ with "Grad" and trigger the bound rule. As a result, instances of "Advising" whose "Grad" attribute values correspond to "this" will all get deleted. Note that the object-oriented paradigm described above is general enough to handle any user-defined new association types without having to be modified.

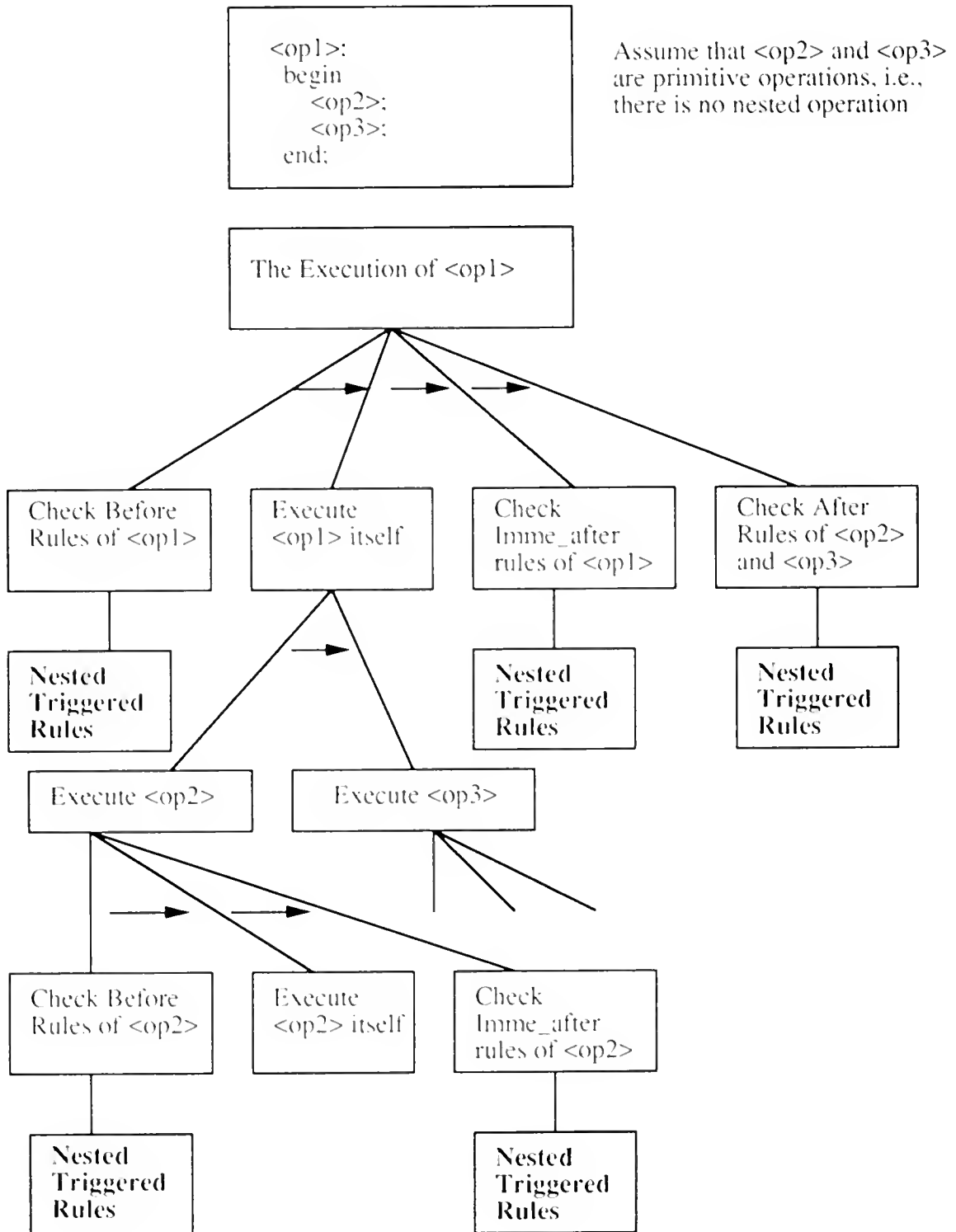


Figure 6.1 Nested Transaction Tree for a Method Invocation

CHAPTER 7

KBMS-SUPPORTED EVOLUTIONARY PROTOTYPING

Along with the development of K, we have also designed a software development methodology, which is based on a knowledge-base modeling approach to evolutionary prototyping. In section 7.1, we give an overview of the KBMS-supported evolutionary prototyping methodology. Section 7.2 describes the knowledge model constructs for the modeling of methods.

7.1 Overview

The development of complex software systems is a costly endeavor. If prototypes can be rapidly constructed to test the structural and behavioral properties of these systems as the developers gain more knowledge about their requirements, then complex systems can evolve from a series of prototyping efforts [BAL82, BOE88]. In this dissertation, we take a knowledge-base modeling approach to evolutionary prototyping of software systems by treating each prototype system as a high-level executable model of the target system. The executable model defines the structural and behavioral properties of the target system at any level of abstraction as desired by the prototyper. It evolves gradually through a series of schema modifications and refinements to provide more

and more details about the requirements and implementations of the target system. At each stage of evolution, the model (i.e., the prototype) can be executed to test its functionalities and performance. All the debugging, modification, and maintenance can therefore be performed directly against the executable model throughout the software lifecycle as shown in Figure 7.1.

As we all know, all software systems are computer programs and, based on Wirth [WIR76] and Kowalski [KOW79], we have the following formula: "Program = Data Structure + Logic + Control." If a knowledge model can uniformly model all types of software systems in terms of their (i) structural properties (corresponding to the data structure aspect of a program and the control structure among program segments), (ii) operations/methods (corresponding to the procedural semantics of program algorithms), and (iii) knowledge rules (corresponding to the declarative semantics of program logic and control), then any software system in the traditional concept can be evolutionarily modeled by this knowledge model throughout its software lifecycle. Note that, although the semantics represented by rules can be implemented in methods, high-level declarative rules make it much easier for a prototyper to clearly specify the semantics instead of burying the semantics in application codes and thus simplify the tasks of testing, modification, debugging, and maintenance. It is not necessary to make the traditional distinction among

software systems (e.g., application systems, operating system, and DBMS) because all of them are executable models of the underlying object-oriented knowledge base. The structural and behavioral properties of all object classes that model programs and application domain objects can be shared and reused among the users.

We have extended an object-oriented semantic association model OSAM* [SU86,89, YAS91] with control associations (which are used for explicitly modeling the control structures of method implementations) as an extensible framework for KBMS-supported evolutionary prototyping. The advantages of this approach are three-fold. Firstly, by using a single unified knowledge model and schema notation, we eliminate the mismatch between the traditional data-oriented models [CHE76, HUL87] and the process-oriented models [NAS73, DEM78, PET81, MIL88] to support both structural and behavioral prototyping within an object-oriented framework. Secondly, all types of software systems, application domain objects that these systems deal with, and related meta information can be uniformly modeled by the knowledge model and managed and processed by an underlying KBMS that uses this knowledge model as its underlying model. Thirdly, instead of serving as throw-aways or being limited to conceptual design, the model of a target system can evolve from specification to implementation throughout the software lifecycle as shown in Figure 7.1. In the KBMS-supported software development system mentioned in

Chapter 1, the evolutionary prototyping process consists of three iterating phases as described in the following.

(1) Analysis/Modeling phase. The prototyper uses the knowledge abstraction mechanisms provided by the abstraction layer to model a software system along with the application domain objects that this system deal with and related meta information at certain level of abstraction as a user-defined schema. Note that since everything is uniformly modeled as objects including the method implementations, this phase actually includes both the specification and implementation of the prototype system. For each method, as the prototype system gradually evolves toward the final system, it is up to the prototyper to decide whether to either (i) model it as a set of object classes along with the structural and control associations among them as will be described in Section 7.2 by using some graphic schema design tool, (ii) simulate it by writing a simple K program to perform some input-output lookup table or to inquire the user, or (iii) actually implement it by writing K codes. Since the structural and behavioral relationships among system components are explicitly specified in the conceptual model of the system being developed, they can be queried and accessed by the prototyper as knowledge-base accesses. For each executable system being modeled, the prototyper also has to write a program definition statement as mentioned in Chapter 5 to serve as the starting point of execution. The program identifier will serve as the name of

the system, which can be used by the prototyper to activate this system in phase (3).

(2) Compiling phase. A K program is translated by the K compiler into executable codes that may contain some external function calls to the KBMS modules. At the same time, the K compiler will also generate instances of system-defined classes "Class," "Association," "Method," and "Rule" as described in Section 3.2 and store these information into the KBMS dictionary. If the compile phase succeeds, the prototyper can proceed to the next phase; otherwise, he/she will go back to the modeling phase to make the corrections.

(3) Testing phase. The prototyper can now test the system that has been compiled in Phase (2) by just typing in the name of the system to run the executable code. The program could ask the user to provide some test data to perform certain computation, and the test result could be stored into the persistent knowledge base. After the Testing Phase is finished, the prototyper can go back to the Analysis/Modeling Phase, retrieve those classes that he/she wants to made modifications from the data-base, modify the executable model of the prototype system, and repeat this process so that the prototype system can gradually evolve toward the final system. Through an iterative process of modeling the system, executing the model for testing its functionality and performance, inquiring the structural and behavioral relationships of system components and modifying the model as more knowledge

about the system has been gained, a series of prototypes will be generated and tested until the final target system (the last evaluated prototype) is derived and released. The conceptual model derived at any stage is executable since the operations and their corresponding codes associated with the object class that model the system components are executable. Note that the above process is totally under the control of a KBMS because everything including the prototype system written in K are treated as objects and can be stored in the persistent data base. The above process also provides a clear distinction between each software development phase and naturally matches with the evolutionary approach of developing complex software systems.

7.2 Method Model and Control Associations

7.2.1 Method Model Concept

In the traditional object-oriented programming, a method consists of a signature that specifies the name of method, parameters, and the data type of a returned value (if a value is to be returned) and the actual program codes that implement the method. However, in the prototyping of a complex system, the prototyper may want to avoid the actual coding of a method at a particular point in time and use instead some simpler table lookup codes to simulate the function of the method (i.e., given some legitimate input data, produce some

legitimate output value by a table lookup). Or the prototyper may feel that the method is still too complicated to code directly and wants to decompose its implementation into program segments interconnected by a control structure. In this case, the method implementation can be represented by a control structure of its program segments that are modeled as object classes with their own methods to define their functionalities. In other words, each program segment (whose size could vary from thousands of statements to a single statement) can be modeled as an object class along with a method (the default method name is "main") to represent the functionality of this segment. To activate a program segment, one just sends a message to an instance of the corresponding object class to invoke the proper method. Through this decomposition process and, at each step, each method associated with an object class is either represented by an actual or simulated program, or by a control structure of program segments that model the method. The model of the entire software system is executable and testable and can gradually evolve into the target system by modifying and refining the executable model. Thus, procedural abstraction and functional decomposition are also incorporated into the proposed object-oriented framework. For the above reason, the meta model of the "Method" class shown in Figure 3.2(b) consists of an execution mode and a signature of its method name, parameter declarations, and the return type. Based on

the execution mode that is either "model" or "operational," the system can choose one of the following to execute: (i) a method_model_object that is the prototype model (schema) of a method implementation, and (ii) a piece of simulated codes or actual implementation of the method in some programming language.

7.2.2 Control Associations

It is shown in Davis [DAV58] that three forms of control structures (sequence/parallel, choice, and repetition) can be used to define all partial recursive (i.e., computable) functions. As mentioned in Section 3.2, one of the advantages of the extensible kernel model is that we can extend the model itself by introducing new association types to carry whatever information we need in association links. In order to explicitly model method implementations in an object-oriented framework, we define a class called "Control" as a subclass of "Association" to model the control relationships among program segments that implement the method. Control associations are categorized as "Sequential" (S), "Parallel" (P) , "Synchronization" (Y), "Testing" (T), and "Context_Looping" (L) as shown in Figure 3.2(a). A method_model is defined as a class schema in which one uses object classes to model program segments and control associations among these object classes to model the control structure of these program segments that implement a particular method.

Figure 7.2 represents some program segments with basic control constructs using control associations. Each rectangular node shown in Figure 7.2 is an entity class that models a program segment that constitutes a method implementation, and each control association in our model represents a possible control flow in terms of message passing between these object classes. A Context-Looping association is used to model the context looping statement in which the system (i) first establishes a relation representing a sub-knowledge-base satisfying the intensional association pattern that is modeled by the first class in Figure 7.2(8), and (ii) performs certain operation for each tuple of the relation as described in Section 3.4. To sum up, each program segment in a method model can be described by a tripple (C, M, P) where "C" is an object class, "M" is the "main" method of "C" that performs the functionality of the program segment being modeled, and "P" is a message passed to a specific instance of "C" to invoke method "M." In other words, "C" and "M" can be thought of as the object-oriented procedural abstraction of the program segment being modeled, and "P" represents the activation of this particular program segment. Through an iterative process, any complex software system can be modeled to any level of details at which point the prototyper can begin to write actual codes in the target language.

The advantages of using method models are four-fold. Firstly, instead of visualizing each method as a black box,

a method model provides a graphic representation of method implementation to capture the behavior properties of a method. Secondly, the method associated with a class that models a program segment can be further modeled by another method model. The process can be repeated to any level of abstraction as desired by the prototyper. The lowest level methods are executable codes. Thirdly, a KBMS can use method models for an automatic generation of codes in the target language where each program segment modeled by (C,M,P) will be replaced by the actual codes of "M" or the actual codes recursively generated from the method model of "M." The resulting codes can then be compiled by the compiler of the target language for execution. Fourthly, a KBMS can directly execute a method model by using an interpreter to dynamically activate each program segment in a control structure following the control association links. Since all the structural and behavioral information needed for execution are stored in the control association links, the execution of a method model can be thought of as the processing of the set of control association links that constitute the method model.

Structurally, each control association link can carry different behavior information as defined by the following attributes where (1) "context_branch" and "sub_kb" are defined by "Context_Looping," and (2) "test_branch" is defined by "Testing" as shown in Figure 3.2(b).

(1) context_branch and sub_kb. a Context_Looping association can be specified by a context_branch attribute whose value could be either "next" or "exit" to represent the iteration or exit of the looping, respectively). Note that the defining class of a Context_Looping association corresponds to the program segment that, when activated, will generate a relation representing the sub-knowledge-base satisfying an intensional association pattern. During the execution of a Context_Looping association, the system will also keep a pointer to the relation (the value of "sub_kb") over which the context looping is performed.

(2) test_branch. a Testing association can be specified by a test_branch attribute whose value could be either "true," "false" (for modeling the "if-then-else" statement), "otherwise," or any other value (for modeling the "case" statement) as shown in Figure 7.2. The defining class of a Testing association corresponds to the program segment that can be activated to generate the proper value of "test_branch" based on which the system can choose one of the possible control flows to follow during the execution time.

The behavioral properties of each control association type are described by the following algorithm of execution. We assume that for each process (in the case of concurrent system) created by a user session, there is a "wait_set" for recording those control association links that are waiting for synchronization. We also assume that each entity class

that models a program segment defines a method called "main" to represent the functionality of this program segment. To activate a program segment modeled by class "X," we create a transient instance of class "X" and apply the "main" method of class "X" to this instance.

Case 1. There is a Sequential (S) association link L1 between class "A" and class "B." We activate the program segment modeled by class "A." Then, if there is no control association link starting from class "B," then we activate class "B" and terminate. Otherwise, we continue to process the next control association link(s) from class "B" (i.e., those control association objects whose defining class is class "B").

Case 2. There is a list of Parallel (P) association links between class "A" and class "B1," "B2," ..., and "Bn." We first activate the program segment modeled by class "A." Then, we fork n new processes in parallel, one for each class "Bi." For each class "Bi," if there is no control association link starting from "Bi," then we activate "Bi" and terminate the process. Otherwise, we continue to process the control association link(s) starting from class "Bi."

Case 3. There is a Synchronization (Y) association link between class "B" and class "A1." There is also a set of Synchronization association links from class B to classes "A2," "A3," ..., "An." Let L1, L2, ..., Ln represent these Synchronization association links, respectively. We first

activate the program segment modeled by class "A." Then, if "wait_set" already contains L2 to Ln, then the synchronization condition is met and we do the following: (1) remove L2 to Ln from the "wait_set," (2) if there is no control association link starting from class "B," then activate the program segment modeled by class "B"; otherwise, continue to process the control association link(s) starting from class "B." Otherwise ("wait_set" does not contain all L2 to Ln), we terminate the process that currently executes L1, and add L1 into the "wait_set."

Case 4. There is a list of Testing (T) association links between class "A" and class "B1," "B2,"..., and "Bn." We first activate the program segment modeled by class "A" and, based on the returned value, the system will choose one Testing association link whose "test_branch" attribute value is equal to either (i) the returned value, or (ii) "otherwise" if none of the test_branch values matches the returned value. Assume this chosen association link is defined from class "A" to class "Bi." If there is no control association link starting from class "Bi," then we activate the program segment modeled by class "Bi" and terminate. Otherwise, we continue to process the next control association link(s) starting from class "Bi."

Case 5. There are two Context_Looping (L) association links between class "A" and class "B1" and "B2." Let L1 represent the association link whose "context_branch" attribute value is "next" (and assume L1 is defined from "A"

to "B1"), and L2 represent the association link whose "context_branch" attribute value is "exit" (and L2 is defined from "A" to "B2"). If "L1.sub_kb" is null, then we activate the program segment modeled by class "A" and return a pointer to a relation representing the sub-knowledge-base over which the looping will be performed. The "sub_kb" attribute of L1 will be set to this pointer. If "L1.sub_kb" points to an empty relation or all the tuples have been processed, then we do the following: (1) delete the relation, (2) set "L1.sub_kb" to null, (3) if there is no control association link starting from "B2," we activate the program segment modeled by "B2" and terminate; otherwise, we process the next control association link(s) starting from "B2." Otherwise (i.e., there are more tuples to be processed), we get the next tuple and continue to process the next control association link(s) starting from "B1."

As shown in Figure 3.2, "Method_model" is a subclass of "Schema." Each method_model object represents the executable model of a method and is described by (i) a set of class association objects (inherited from class "Schema"), (ii) a starting point that is a control association object in (i), and (iii) a set of local variable declarations. Note that in order to unambiguously preserve the semantics of the order of execution (control flow) when a method model is mapped into a set of association objects, each class that appears in more than one places in the method model must be recorded by using

alias names internally. An alias name is formed by appending an underscore and an integer to the class name, e.g., Sort_1 and Sort_2. Note that without using alias names, the system will not be able to restore the model correctly. For example, the control structure restored from three consecutive "Sequential" associations A-B, B-C, and C-A will form an infinite loop instead of a sequence if no distinction is made between these two appearances of class "A." Besides, each method_model object must know which association object is the "starting point" of execution. From the starting point, the method model can be restored and processed by following the control associations.

In the following, we illustrate the concept of evolutionary prototyping and the use of all types of control associations by developing "eval_GPA()", which is a method of Student as shown in Figure 3.1. Note that although "eval_GPA()" is a rather simple method that normally could have been directly coded, the technique illustrated by this example can be applied to the modeling of complex methods of a large software system to any level of details. Assume we have defined Transcript as an entity class whose each instance represents the grade point of a particular student for a particular course, and we need a program to compute the GPA of a given student. For this example, it is obvious to model this program as a method of class Student with the signature "eval_GPA() : GPA_Value." In the beginning, one might just

write a simple piece of simulated codes to generate some legitimate GPA_Value from some given legitimate student instance as the receiver of this method by either performing some table lookup or inquiring the user interactively so that this method can be executable (in operational mode).

Later on, one may decide to model the detail of this method by decomposing its functionality into five consecutive program segments: (1) compute the total grade points of this student and assign this value to a local variable, (2) compute the total credit hours of this student and assign this value to a local variable, (3) get the GPA by dividing results from (1) and (2), (4) print a message if the GPA is below 2.0, and (5) return the GPA. Each program segment can be modeled as an entity class with a "main" method to represent its functionality, and each "main" method can be given simulated or actual codes or recursively modeled as described in Section 7.1. Note that for this example, both segment (1) and segment (2) can be further decomposed as Context_Looping control structures, and the computations can be performed over the same context concurrently. Therefore, we combine them together to illustrate the Context_Looping, Parallel, and Synchronization associations. A method model at a particular stage of decomposition is shown in Figure 7.3. We first declare local variables s1, s2, and GPA to hold the accumulated grade points, accumulated credit hours, and GPA value, respectively. The receiver of this method is denoted

by the pseudo variable "this." We first use a Context_Looping association to iterate over the context specified by "this *<[student] t:Transcript *>[course] c:Course" to evaluate the accumulated grade points and credit hours of this student in parallel. Note that the updates of s1 and s2 are performed in parallel and must be synchronized before the execution can be continued. After the looping is finished, we get the GPA value by dividing s1 by s2. A message is printed if this student has a GPA lower than 2.0. Finally, we return the GPA value. Note that in some cases it is necessary to introduce entity classes that model null program segments in a control structure. For example, the classes "null_1" and "null_2" in Figure 7.3 model the null program segments that serve as the "fork" and "join" points of control flows, respectively. This example shows that it is possible to model a method recursively to such a detailed level that each program segment contains only a single statement. Naturally, the segment size (i.e., the level of detail) in a model will be determined by the prototyper. By using a graphic user interface as part of a prototyping environment, a prototyper can click the mouse button to select any class in a method model and view the program segment it represents as shown in Figure 7.3. A control structure of the kind shown in Figure 7.3 can be used by a KBMS to dynamically execute a method model or automatically generate the proper

executable code that implements the method as shown in Figure 3.1.

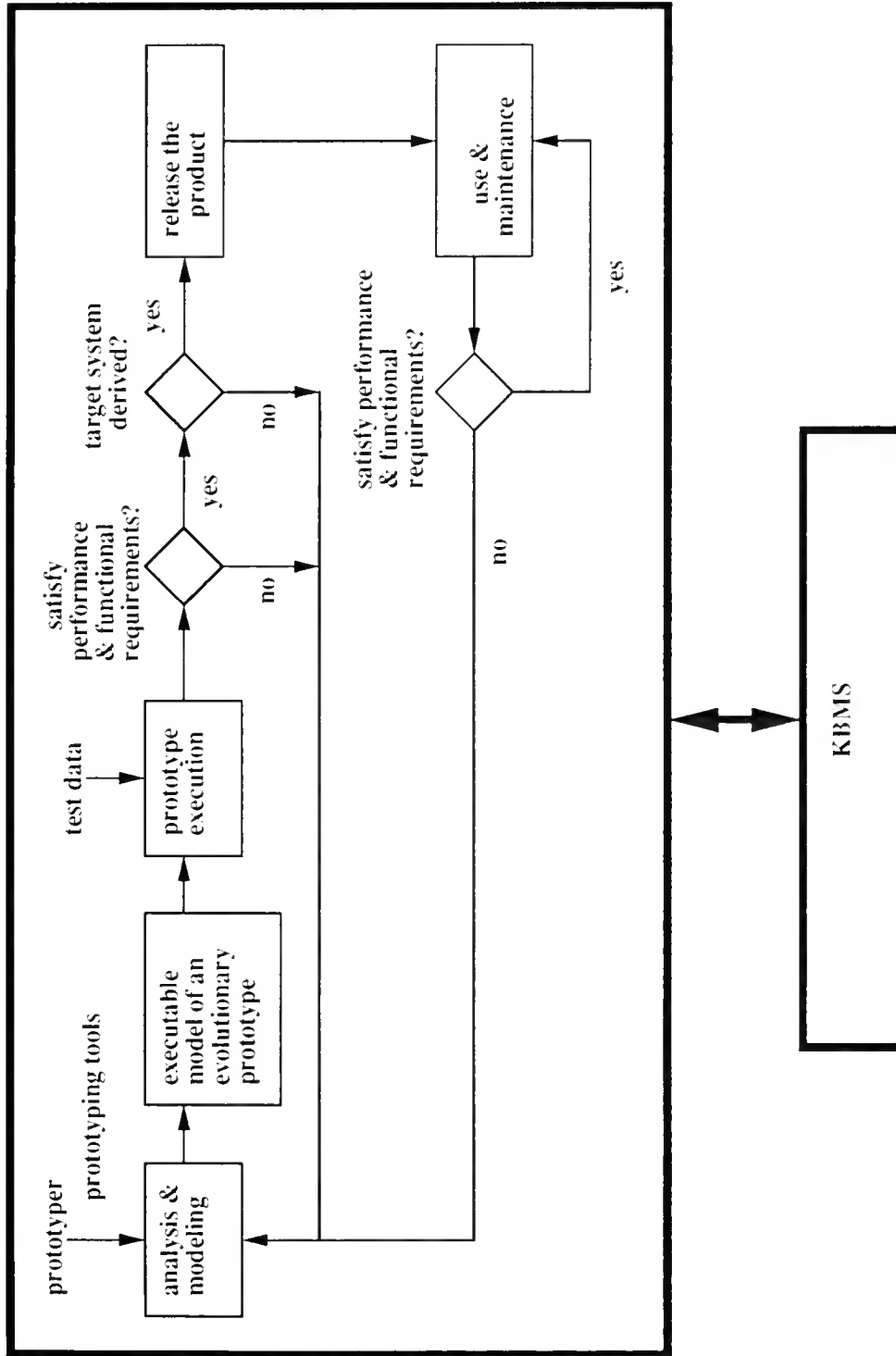


Figure 7.1 An Overview of a KBMS-supported Evolutionary Prototyping Process

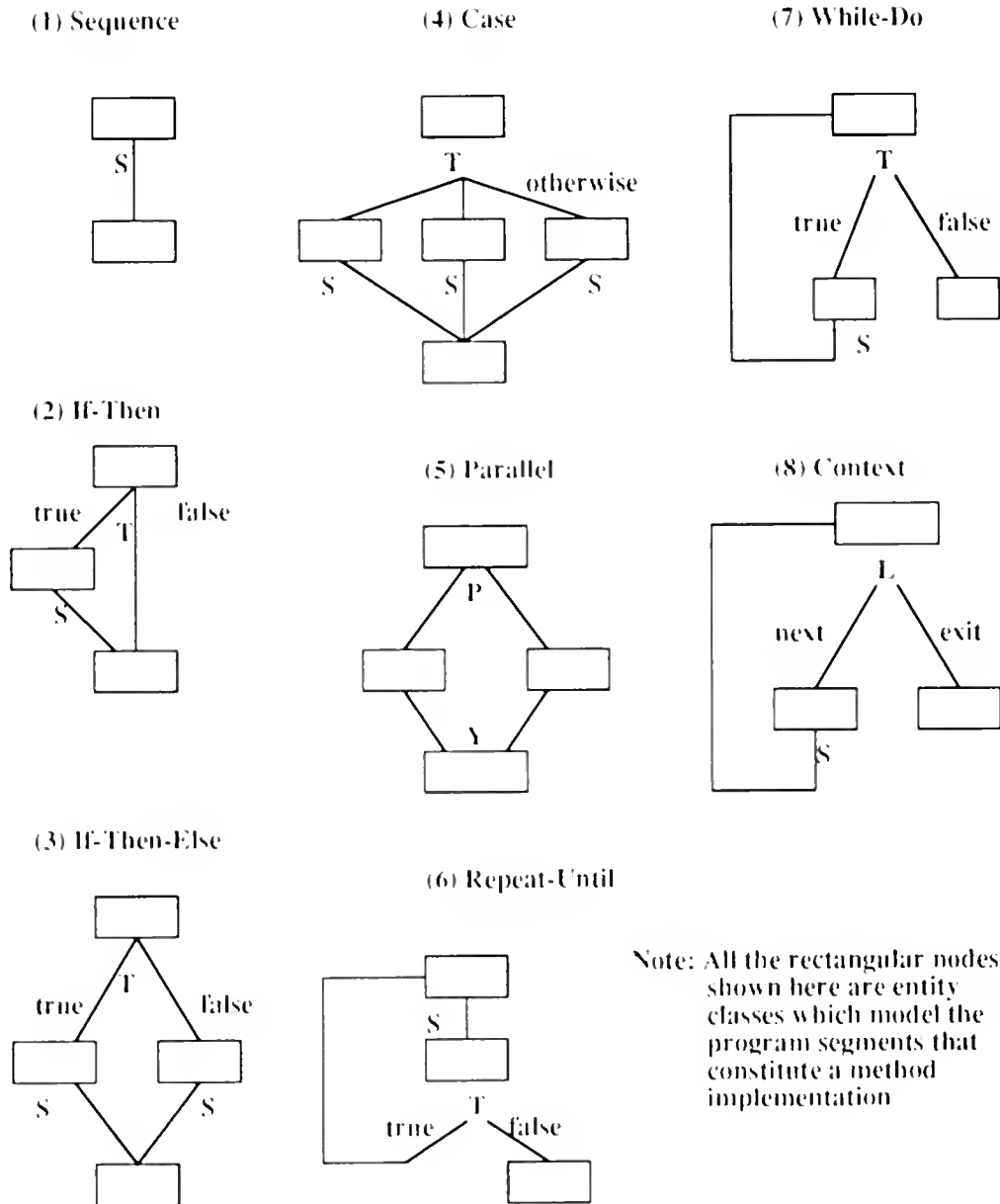


Figure 7.2 The Modeling of Basic Control Constructs

Student::eval_GPA(): GPA_Value

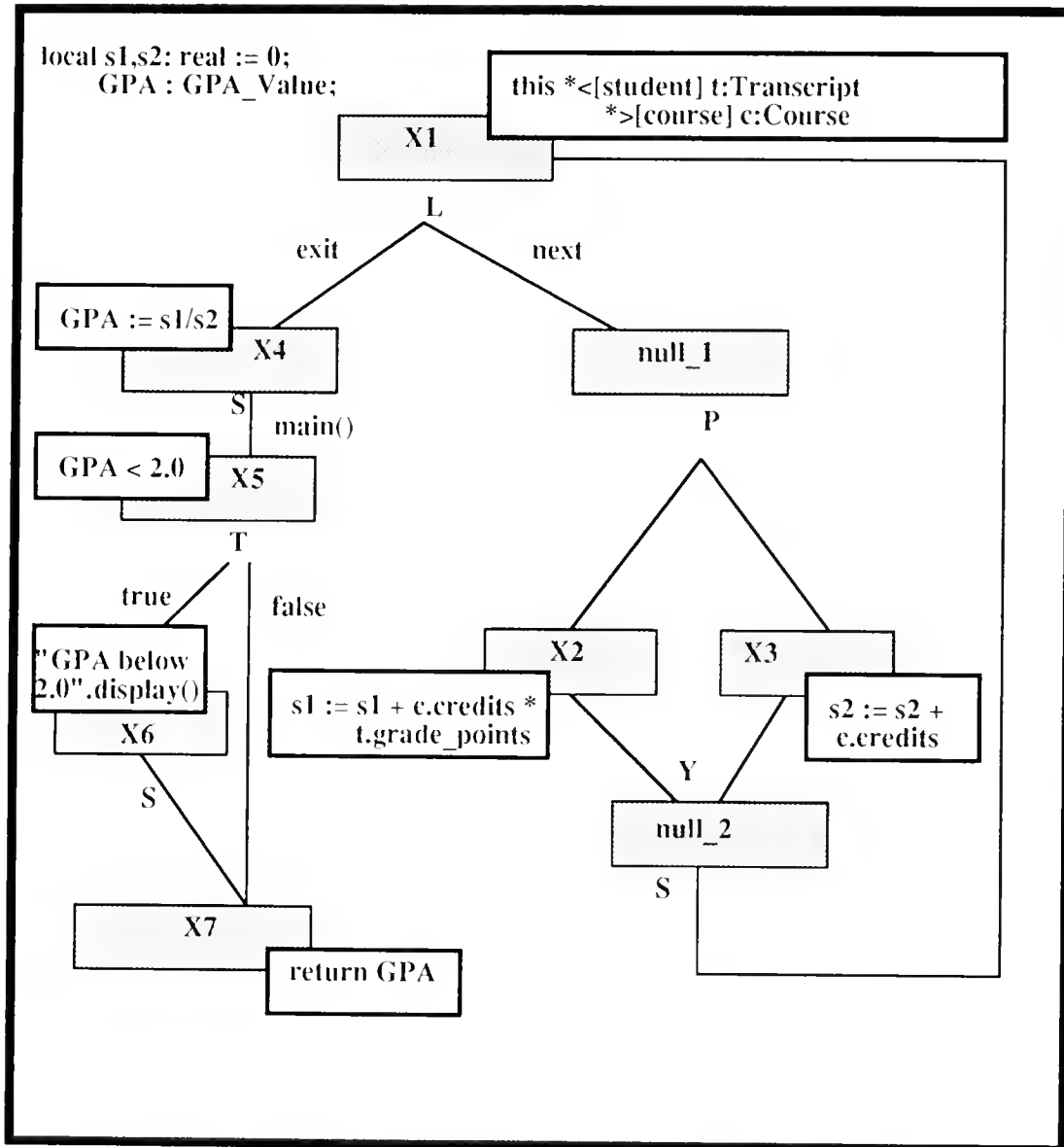


Figure 7.3 The Method Model of "eval_GPA()"

CHAPTER 8 SYSTEM ARCHITECTURE AND IMPLEMENTATION

8.1 System Architecture

In order to support the software development and prototyping activities described above, we have designed an open, modular, and extensible architecture based on the layer structure of a universal KBMS described in Chapter 1. An overview of the system architecture is shown in Figure 8.1. Note that each major component of the evolutionary prototyping system is represented by a handler class for better modularity, and we use the "Using" association to represent the client-server relationship among these components. The K_Handler module, the KBMS_Handler module, and Storage_Handler module correspond to the language layer, control layer, and storage layer of a universal KBMS, respectively. We have implemented the K_Handler and KBMS_Handler on Sun 4 in C++. The Storage_Handler is currently supported by ONTOS 2.1 [ONT91], a commercial object-oriented data-base system for low-level object management and transaction management. The Prototype_Handler will be built on top of the K_Handler as an extension of this project. The functionalities of each module is described as follows:

8.1.1 K_Handler

This class is responsible for compiling and executing any K program. The interface of K_Handler consists of "compile," and "execute" operations. Given a file containing a K program, the "compile" operation will call the parser class, which is the top-level module of K_Handler, to do the compilation in the following two phases.

Phase 1. First, the parser will call the lexical analyzer to read the input K program and return a token stream, based on which the parser will perform syntax checking according to the grammar rules of K. At the same time, the parser will call the intermediate code generator to do the following:

(1) Generate meta information as instances of the kernel classes (such as "Class," "Method," "Rule," and "Association") and call the KBMS_Handler (which in turn will call the Dictionary_Handler) to insert these objects into the knowledge-base dictionary.

(2) Generate an intermediate C++ program file (.cxx) for each class definition, which contains the corresponding C++ codes for methods and rules of each class definition.

(3) Generate an intermediate program file (.cxx) for each program definition, which contains the corresponding C++ "main" program.

Note that if a class has already been defined, then all the meta-information about the old definition will be deleted

from the dictionary. Since all the semantic information is stored in the dictionary, semantic checking facilities are built into Dictionary_Handler and will be invoked by the intermediate code generator (through the KBMS_Handler) when instances of the kernel classes are inserted, deleted, or updated, and when C++ codes for K methods and rules are generated. Note that as a strongly typed language, K can detect most of the logic errors as type errors at compilation time. At the end of the first phase, if any error has been reported, then the parser will abort the "compile" operation by undoing all the changes to the dictionary and deleting all the intermediate C++ files. Otherwise, the parser will proceed to the second phase.

Phase II. Based on the meta information stored in the dictionary from phase I, the parser will call the intermediate code generator to do the following:

- (1) Generate the C++ header file (.h) for each K class that has been compiled in phase I.

- (2) Append to each C++ program file (.cxx) generated from phase I the necessary C++ methods that are required by ONTOS.

The parser will then call the machine code generator to generate the machine codes. The machine code generator will do the following:

- (1) Run the "cplus" (persistent C++ compiler) utility of ONTOS to compile the C++ program file and header file for each class definition and generate a corresponding .o file.

(2) Link the .o files of all the necessary classes with the KBMS_Handler module and Storage_Handler module (ONTOS library), and finally generate an executable file for each program definition.

(3) Run the "classify" utility of ONTOS to update the ONTOS data-base schema at the storage level.

Note that after compilation, for each K program definition, there will be an executable file with the same name as that of the program. To activate a compiled K program, one can send the "execute" operation to the K_Handler via a graphic user interface, or just type in the program name at the UNIX command shell. Note that while the K_Handler makes a lot use of the KBMS_Handler to support semantic checking and code generation, the executable codes generated by the K_Handler also contains many function calls to the KBMS_Handler to support persistent/transient objects, pattern-based queries and quantifiers, and rule-based computations. An overview of the program development process is shown in Figure 8.2.

8.1.2 Prototype_Handler

This class is responsible for processing Method_Model objects described in Chapter 7. The interface of Prototype_Handler consists of a single "translate" operation. During the compilation of a K program, if the execution mode of a method is specified as "prototype," the K_Handler will

send the "translate" message to `Prototype_Handler` that will retrieve the corresponding `Method_Model` object of this method from the dictionary (by calling the `KBMS_Handler` that in turn calls the `Dictionary_Handler`) and translate it into corresponding K codes as the return value to `K_Handler` for compilation.

8.1.3 KBMS_Handler

This class provides all the KBMS facilities to its clients. As the top-level interface class of all the KBMS components, `KBMS_Handler` hides all the details of its constituent classes and serves as the controller of all incoming messages by dispatching them to appropriate KBMS component classes that actually implement the corresponding methods. `KBMS_Handler` uses the following classes:

(1) `Object_Handler`, which performs the basic KBMS operations such as "new," "pnew," "insert," "delete," and "destroy" described in Section 3.5.

(2) `Query_Handler`, which performs the basic link operations "associate" and "non-associate" and returns the corresponding contexts as tables of extensional association patterns as described in Section 4.2.

(3) `Rule_Handler`, which performs the triggering of applicable rules as described in Section 5.2.

(4) `Dictionary_Handler`, which serves as the interface of the kernel schema described in Section 3.2 and provides such

functionalities as "getNpar" (get the number of parameters of a given method), "matchParam" (check the type compatibility between a formal parameter and an actual parameter of a given method), and "getDefClass" (get the defining class of a given method), etc.

(5) Transaction_Handler, which performs "startTransaction," "commitTransaction," and "abortTransaction" operations.

Note that Dictionary_Handler is used by K_Handler for establishing semantic information, performing semantic type checking, and generating intermediate C++ code. Transaction_Handler is used for handling high-level transactions (i.e., transactions from the K_Handler's point of view) and mapping these high-level transactions to the storage layer system-dependent transaction calls.

8.1.4 Storage_Handler

This class provides the low-level storage management and transaction management (concurrency control and recovery) facilities to the KBMS_Handler.

8.2 Implementation: Mapping from K.1 to C++

A prototype implementation of K_Handler, KBMS_Handler, and a version of K (K.1) has been implemented on Sun4 using Ontos C++ [ONT91]. All the linguistic facilities described in previous Chapters except control associations have been

implemented. The only limitation is that in our current implementation, we treat the execution of each K.1 program (instead of each nested K.1 method) as a single transaction because the nested transaction model is not well supported in the `Storage_Handler` level. Therefore, the execution of any "abort" statement will actually abort the whole transaction. The specialized tools LEX and YACC are used to generate the lexical analyzer and parser. The `Storage_Handler` is currently supported by ONTOS 2.1, and the `Prototype_Handler` will be added in the future. The overhead of using a commercial OODB (such as ONTOS) is that while we use only a small portion of its functionalities for low-level storage and transaction management, we have to link with its entire system codes that occupy lots of memory space. Later on, we will consider to implement our own low-level storage and transaction management facilities. In this section, we give an overview of the implementation in terms of the mapping from K.1 to C++. As a naming convention, any identifier used internally by `K_Handler` will start with "_K." A detailed description of the implementation can be found in Arroyo [ARR92].

8.2.1 Object and Instance

As described in Chapter 3, entity class objects are referred at the instance level by their iids, and domain class objects are directly referred by their values. At the C++ level, each iid is mapped into a pointer to an instance of

system-defined class "_KInstance." Each _KInstance holds such information as oid, cid, and pflag (a flag indicating that whether this instance belongs to a persistent EClass_Object), and serves as an abstract reference to a K.1 instance. To access a K.1 instance, the system uses the "cid" and "pflag" of a _KInstance to get the proper object table, and uses the "oid" as an index to retrieve the actual K.1 instance. At the C++ level, any single-valued entity class instance variable or reference attribute will be uniformly converted to a pointer of _KInstance, and any single-valued domain class variable or value attribute will be converted to a pointer of the corresponding C++ class. Note that primitive domain classes (integer, real, character, string, and boolean) are re-implemented on top of their corresponding C++ primitive types so that one can define specialization or any other type of association for a domain class.

8.2.2 Schema

Each schema (a set of class associations) defined in K.1 is mapped into the kernel class structure described in Chapter 3, which in turn is mapped into a C++ class structure at the implementation level. The mapping of various class association types is described as follows:

Generalization. As C++ does not support the distributed storage model of entity class objects described in Chapter 3, inheritance for entity classes is supported by the object

manager rather than C++. At the C++ level, any inheritance lattice of entity classes defined in K.1 is flattened so that all the entity classes (including system-defined kernel entity classes) become immediate subclasses of "EClass_Object." At the same time, every entity class at the C++ level defines a pseudo attribute "_KSUP<class_name>" or "_KSUB<class_name>" for each of its immediate superclasses or subclasses, respectively. The sole function of such an attribute is to allow the object manager to efficiently identify the superclass or subclass of a given object instance by extracting information from the attribute name. For example, the following C++ codes will be generated in the "Student.h" file after the Student class defined in Figure 3.1 is compiled. For domain classes, their inheritance lattices are directly mapped into C++ class structures without being flattened, and no such pseudo attributes are generated.

```
class Student : public EClass_Object
{ public:
    _KInstance* _KSUPPerson;
    _KInstance* _KSUBGrad;
    ...
};
```

Aggregation. As mentioned above, single-valued reference attributes and value attributes are converted to pointers of _KInstance and corresponding C++ classes, respectively. On the other hand, multi-valued attributes and variables are mapped into pointers of container classes, which are system-defined

domain classes `_KESet`, `_KEList`, `_KEArray`, `_KDSet`, `_KDList`, and `_KDArray`. Each container class defines an attribute "elementType" to store the type information for each of its instances. Besides, for each reference attribute, the system automatically defines and maintains an inverse link in the constituent class of that aggregation association. Each inverse link is named as "`_KINV<defining_class>_<link_name>`," and is used for supporting referential constraint and bi-directional navigation. Note that the inverse link for a single- or set-valued attribute will be set-valued, and the inverse link for a list- or array-valued attribute will be list-valued. For example, the following C++ codes will be generated in the "Student.h," "Course.h," and "Department.h" files after class Student defined in Figure 3.1 is compiled.

```

class Student : public EClass_Object
{ public:
    _KESet* enroll; /* K: enroll : set of Course */
    _KDArray* college_report;
                /* K: array[4] of GPA_Value */
    _KInstance* major; /* K: major : Department */

    ... /* possible inverse links for attributes
        defined by other entity classes */
private:
    S#_value* S#; /* K: S# : S#_Value */
    ...
};

class Course : public EClass_Object
{ public:
    _KESet* _KINVStudent_enroll;
    /* inverse link for "enroll" defined by "Student */
    ...
};

```

```

class Department : public EClass_Object
{ public:
    _KESet* _KINVStudent_major;
    /* inverse link for "major" defined by "Student */
    ...
};

```

For any abstract reference (i.e., instance of _KInstance, _KESet, _KEList, or _KEArray) that serves as the value of a reference attribute of an EClass_Object, the system keeps an "owner" attribute as a pointer pointed back to that particular EClass_Object. To update a reference attribute, the system will do the following: (i) remove the inverse links between those instances referred by the attribute and the "owner" of this attribute, (ii) updates the abstract reference itself, and (iii) establish the inverse links between those instances currently referred by the attribute and the owner of this attribute. Similarly, when an instance is added to or removed from a multi-valued reference attribute, the corresponding inverse link of this instance will be automatically updated. All the semantic information of an aggregation association (e.g., the constituent class, and the dimension of an array) are kept in the dictionary for semantic checking. For other types of association (e.g., using and friend), the K_Handler just stores the meta information into the dictionary, and no corresponding C++ code is generated.

8.2.3 Method/Operator Declaration

Method declarations are directly mapped into C++ with the same encapsulation level. The return type and parameter types are mapped into pointers to `_KInstance`, corresponding C++ domain classes, or container classes in the same way as the mapping of attributes and variables described above. Similarly, overloaded operator declarations are mapped into C++ methods with system-defined names. For example, the following C++ codes will be generated for the K.1 method and operator declarations shown as comments.

```
string* test(_KInstance* s, integer* i);
/* K: method test(s: Student, i: integer) : string; */

boolean* _KOPGT (_KInstance* s);
/* K: operator > (s:Student) : boolean; */
```

8.2.4 Method/Operator Body

Each K.1 statement in a method or operator body is mapped into the corresponding C++ codes by the intermediate code generator as follows:

Basic control structures (block-statement, if-then-else-statement, for-statement, while-statement, break, continue, and return) are mapped into their C++ counterpart statements. Local variable declarations are mapped into C++ declarations as described above. Each case-statement is mapped into a sequence of if-then-else-statements. Each context-looping-statement is mapped into a while-loop over a `_KPattern`

instance, which is a table structure representing the sub-knowledge-base returned by the specified association pattern. Object statements (delete and destroy), rule statements (activate and deactivate), and abort statement are mapped into method calls to the KBMS_Handler.

Each K.1 expression is mapped into a sequence of stack operations that are applied to an abstract stack machine implemented in C++ as class `_KStack`. All the operands of an operator, or the receiver as well as all the parameters of a method, are pushed into the stack for evaluation, popped out from the stack after evaluation, and the result of the evaluation is pushed back into the stack so that it can be available as operand, receiver, or parameter of the next expression. At run time, an instance of `_KStack` will be created for each method invocation as the local abstract stack machine. The advantage of this approach is two-fold. First, the code generation is straight forward since we are using a LR parser that also operates on an abstract state machine. Second, we have a uniform evaluation model for all types of expressions (including context expressions). A detailed description of mapping from K.1 expressions to stack operations can be found in Arroyo [ARR92].

Besides, in order to incorporate rule checking, a C++ statement that sends a "beginOp" method to the KBMS_Handler will be generated at the beginning of a method body, and a C++ statement that sends an "endOp" method to the KBMS_Handler

will also be generated at every possible exit point of the method body (i.e., before every "return" statement and at the end of the method body). Similarly, "beginOp" and "endOp" messages will be sent to the KBMS_Handler at the beginning and end of the C++ codes for (i) new, pnew, and insert operations, (ii) delete and destroy statements, and (iii) assignment (update) expressions, respectively. The KBMS_Handler will then dispatch the messages to Rule_Handler that uses a stack to keep track of all the triggering events. An overall structure of a C++ method "test" defined by "Student" is shown below.

```
boolean* test(_KInstance* s, integer* i)
{ _KBMS ->beginOp(..);
  ... /* C++ code generation */
  _KBMS ->endOp();
  return..;
  ...
  _KBMS ->endOp();
  return..;
  ...
  _KBMS ->endOp();
};
```

8.2.5 Program

Corresponding to each K program definition, a C++ "main" program is generated in a .cxx file that uses the corresponding K program name as its file name. The C++ code generation for each K program is similar to that of a K method except that (i) we generate a C++ "main" program with "void" return type, and (ii) we replace the function calls "beginOp" and "endOp" to the KBMS with "startTransaction" and

"commitTransaction," respectively. During the execution of a K program, K_Handler will first send a "startTransaction" message to KBMS_Handler and begins to execute the "main" program. At the end of a user program, a "commitTransaction" message is sent to the KBMS_Handler to commit all the changes to the knowledge base.

8.2.6 Rule

After compilation, the meta information of each rule (e.g., rule name and trigger conditions) are stored in the dictionary, and the rule body (condition-, action-, and otherwise-clause) is mapped into a corresponding C++ system-defined method "_KRULE<rule_name>" of the defining class. This method takes no parameter, and returns an integer indicating whether an "abort" statement is executed in the triggering of this rule. The mapping of the condition-, action-, and otherwise-clause can be regarded as that of an equivalent simple or nested (when the condition-clause is a guarded expression) if-then-else statement. For example, suppose class Student defines a rule "StudentRule1" with (C1 | C2) (guarded expression) in the condition-clause, and an "abort" statement in the action-clause. An overall structure of such a method mapped from "StudentRule1" is shown below. Note that the current version of K (K.1) does not support nested transactions, and the execution of an "abort" statement will actually abort the entire user program. Besides, for these

system-defined methods mapped from K.1 rules, the intermediate code generator does not generate codes for sending "beginOp" and "endOp" messages to the KBMS_Handler because no rule checking is necessary. To trigger the rule "StudentRule1," the Rule_Handler just has to apply the method "_KRULEStudentRule1" to a particular student instance upon which certain knowledge-base event occurs and matches one of the trigger conditions of StudentRule1.

```
int _KRULEstudentRule1()
{ int flag = 1;
  ... /* code for evaluating C1 */
  if (*(boolean*)_KStack.pop()) /* test C1*/
  { ... /* code for evaluating C2 */
    if (*(boolean*)_KStack.pop()) /* test C2*/
    { /* action */
      _KBMS->abortTransaction(); /* abort */
      flag = 0;
    }
    else {...} /* otherwise */
  }
  return flag;
};
```

8.2.7 Bootstrapping

In order to test the functionality of K_Handler, we use K.1 itself to write the Dictionary_Handler (which includes the dictionary schema described in Chapter 3) and bootstrap the system in the following two steps. First, we use the bare K_Handler (without Dictionary_Handler support) to compile the Dictionary_Handler as described in Section 8.1. Note that no dictionary update is performed because it does not exist yet. The resulting .h files are used by the machine code generator

to set up the ONTOS data-base schema for the dictionary. At the end of this stage, we have established an empty dictionary (without even the definition of the kernel classes themselves) that is ready to be populated. Second, we use K_Handler to compile the Dictionary_Handler again. At this time, all the system-defined kernel classes (e.g., class, association, method, rule, and primitive domain classes) are compiled and their meta information are inserted into the dictionary. At the end of this stage, the Dictionary_Handler is fully established and thus the K_Handler is ready to use.

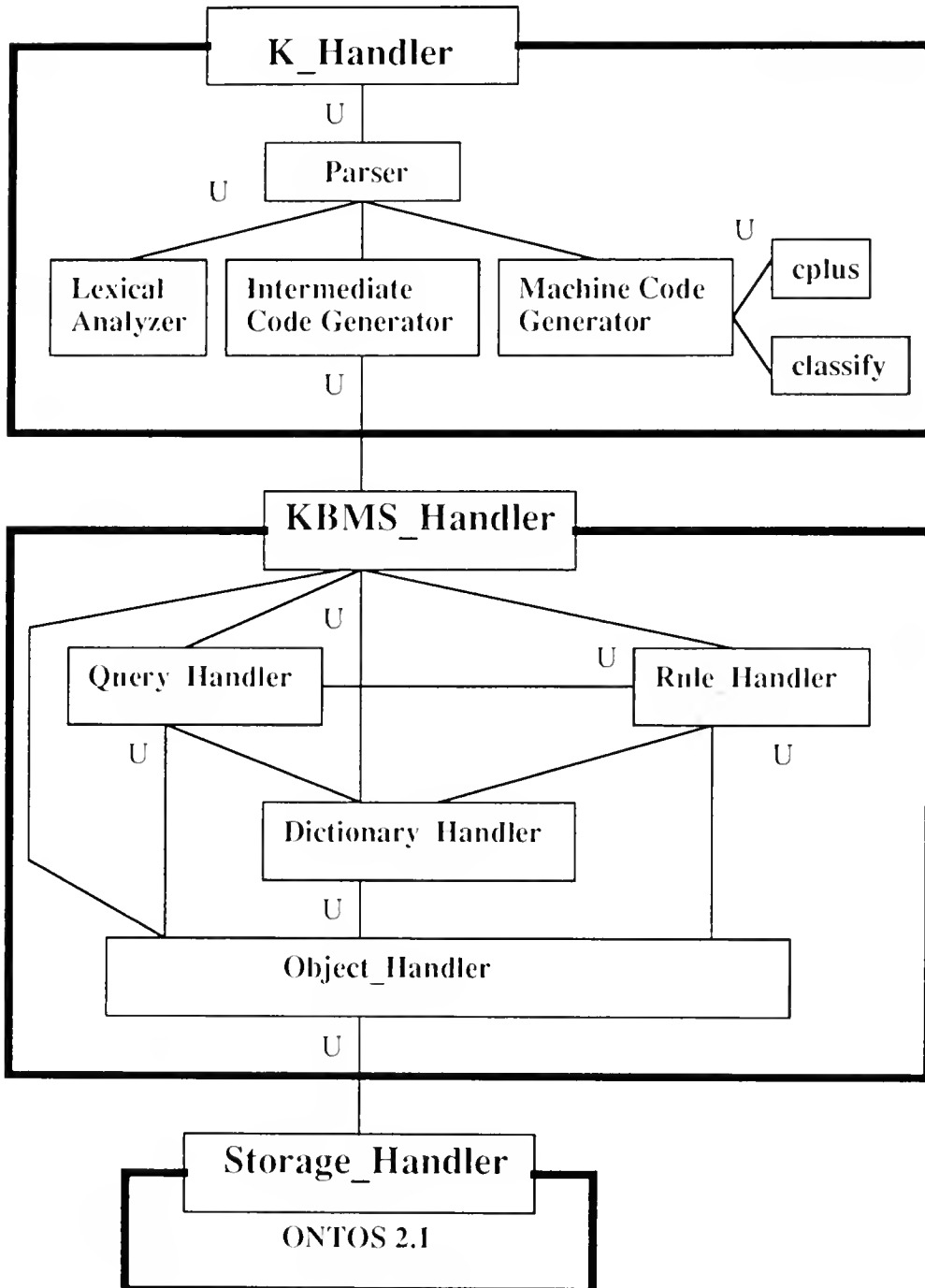


Figure 8.1 The System Architecture of K

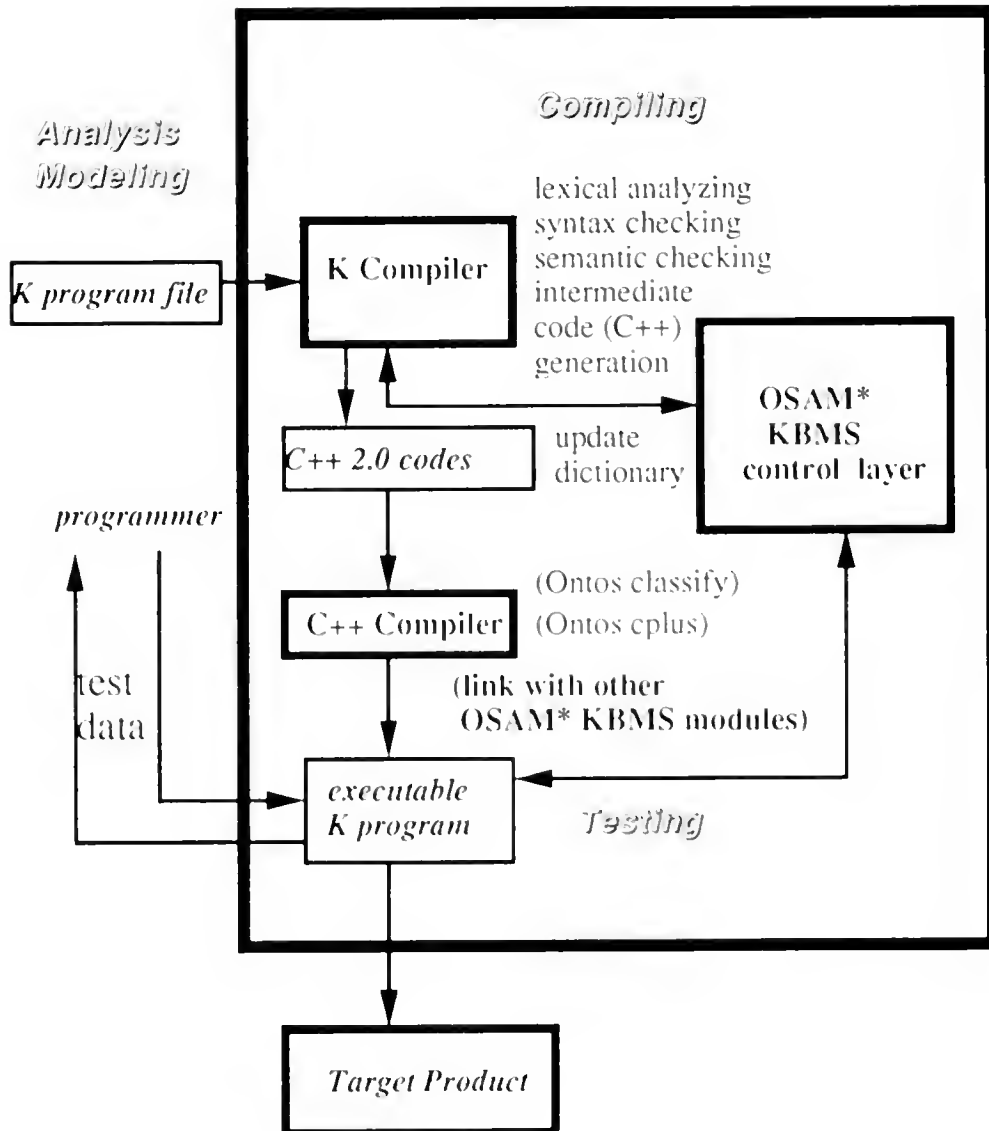


Figure 8.2 Software Development Process Using K

CHAPTER 9 CONCLUSION AND FUTURE RESEARCH DIRECTIONS

9.1 Conclusion

In this dissertation, we have described the design and implementation of an object-oriented knowledge-base programming language K in terms of its (i) underlying knowledge model (abstraction layer), (ii) linguistic facilities (language layer), (iii) implementation architecture (control layer), and (iv) application to evolutionary prototyping of software systems. K serves as a high-level interface of the OSAM*.KBMS knowledge-base management system to define, query, and manipulate the knowledge base, as well as to write codes to implement any data/knowledge-intensive application system. In general, the contribution of this research lies in integrating the techniques introduced in data-base management system, programming language, and software engineering in an object-oriented framework toward a KBMS-supported software development system. Specifically, my contribution to this research is two-fold as described in the following.

First, on the language aspect, I have designed the knowledge-base programming language K as the high-level

interface of the KBMS-supported software development system. Starting from a query language and rule language, K seamlessly incorporates the query processing, rule processing, persistence, and general computation facilities within an object-oriented framework. In addition to such well-known object-oriented virtues as abstract data types, information hiding, complex objects, relationships, inheritance, reusable codes, etc., K provides six important features as follows:

(1) A rich set of knowledge abstraction mechanisms for supporting the underlying OSAM* knowledge model, which captures any application domain knowledge in terms of the structural associations, methods, and knowledge rules.

(2) A strong notation of address-independent object identifiers.

(3) A persistence mechanism for supporting both persistent and transient objects.

(4) A flexible type system that supports both static type checking and multiple views of objects.

(5) A declarative knowledge retrieval mechanism based on object association patterns.

(6) An extended computation paradigm for supporting both procedural and rule-based computations.

I have also participated in the implementation of the first prototype version of K (K.1) and its supporting OSAM*.KBMS as the first step toward a full-fledged KBMS-

supported software development system at the database systems Research and Development Center of the University of Florida.

Second, on the software development methodology aspect, I have presented a knowledge-base modeling approach to evolutionary prototyping of software systems by treating each prototype system as a high-level executable model of the target system. The executable model defines the structural and behavioral properties of the target system at any level of abstraction as desired by the prototyper. It evolves gradually through a series of schema modifications and refinements to provide more and more details about the requirements and implementations of the target system. At each stage of evolution, the model (i.e., the prototype) can be executed to test its functionalities and performance. I have extended the object-oriented semantic association model OSAM* [SU86,89a,b, 92, YAS91] with control associations to explicitly model the control structures of method implementations at any level of detail as desired by the prototyper. The advantage of this approach is three-fold. Firstly, by using a single unified knowledge model and schema notation, we eliminate the mismatch between the traditional data-oriented models and the process-oriented models to support both structural and behavioral prototyping within an object-oriented framework. Secondly, all types of software systems, application domain objects that these systems deal with, and related meta information can be uniformly modeled by the knowledge model and managed and

processed by the KBMS that uses this knowledge model as its underlying model. Thirdly, instead of serving as throw-aways or being limited to conceptual design, the model of a target system can evolve from specification to implementation throughout the software lifecycle. This extension paves the way for a software development system in which the developer can graphically model, query, and execute any software system at any level of abstraction.

To sum up, K is more suitable for the purpose of KBMS-supported software development and prototyping than traditional programming languages because of its (i) modeling capability, (ii) more declarative style of programming, (iii) general computation capability, and (iv) KBMS support for object management and persistence. K serves as a single integrated language for the whole KBMS-supported life-cycle of software development from specification to implementation.

9.2 Future Research Directions

Many research topics under the categories of language extension and prototyping environment can be spawned based on this work. In the following, we list some of the important tasks that should be considered as future research directions.

9.2.1 Language Extension

Generic mechanism and extensibility. Generic, or parameterized, facilities (generic class/method/rule) will be added in the future to support extensibility. They can be implemented as some pre-compilation macros and a pre-processor can be built on top of K compiler to translate generic definitions to bound definitions. The other approach is to extend the execution engine to dynamically handle the bindings at run time.

After the implementation of the generic mechanism is completed, the user can not only add new association types as specialization of class "Association," but also extend the definitions of kernel association classes such as aggregation. For example, an aggregation definition now looks like:

```
my_friends : set of Person where
    [default := <..>]
    [null := <..>]
    [optional := <..>]
    [unique := <..>]
    [unchangeable := <..>];
```

In other words, each link specification can now contain not only the basic definition, but also followed by a where-clause containing the assignments of other aggregation associations defined by the particular association class. The compiler will look into the KBMS Dictionary for these meta-information. Similarly, it is possible to use generic rules to implement the mechanism for supporting optional "user-defined key" for

any entity class, which is suggested in the next-generation data-base manifesto [SIG90].

Primitive types and enumeration. In the later version of K, we will take the approach of implementing primitive domain classes (integer, real, character, and string) as primitive C++ types for better performance. We would also allow the user to define a domain class as the enumeration of a list of values (which in general can be any domain class objects, not necessarily integers as in C++) in the following form:

```
domain_class <class_name> is  
    enumeration of <values>  
end <class_name>;
```

Garbage collection and code optimization. In order to provide reasonable performance, some garbage collection and code optimization facilities must be built into the system. The garbage collection facility will be used to remove unnecessary objects from the main memory and solve the memory fragmentation problem so that the execution of a complex system will not run out of the memory. The code optimization facility will be used to optimize the C++ code generated from K compiler, especially the code for context looping statement by using some query optimization techniques and reducing the redundant efforts of trying to find applicable rules for each looping.

Virtual class and dynamic binding. The current version of K uses static type checking and does not provide the

mechanism for supporting virtual (or abstract) classes and dynamic binding, both of which are useful in object-oriented programming and prototyping. We would like to introduce these features in the future, and replace static type checking with both strong type checking and dynamic type checking.

Rule validation. Current Rule_Handler module does not provide any facility for rule validation. Techniques must be developed to detect any possible cycle and/or conflict in rule definitions.

Coupling modes. Currently, we considered only "before," "after," and "in_parallel" coupling modes. Depending on the availability of more sophisticated transaction manager, more complex ones as suggested in HiPAC [HSU88] (e.g., independent, causality-dependent) could be introduced to provide more flexible and finer grained control over rule execution.

Multi-paradigm computation. Current implementation of K supports object-oriented and rule-based computations. We would like to evolutionarily incorporate parallel, real time, and non-deterministic computations as described in Chapter 6. A more sophisticated transaction manager will be necessary to support such computation activities.

Temporal/historical references and event detection. We would like to add the facilities of temporal and historical references into the language by extending association patterns and rule definitions with such constructs as "before," "after," "at," "during," "every," "within," etc. We will also

extend the rule handler with event detection capability to detect the occurrence of temporal triggering events. Object manager will also be extended to handle historical data.

Versioning and schema evolution. Version control and temporal mechanism for handling historical data will be defined by the system-defined E_Class_Object and inherited by all its subclasses. An entity class object could have multiple versions with distinct vid but the same oid and iid. History reference will keep track of the system evolution and reduce the burden on the prototyper to manage history such as many uses of temporary variables. Versioning is not only a way to implement historical data but also a necessary facility for software development. We would like to extend the language to allow the user to define new versions and call-in/call-out different versions of an object [BEE88]. For a complex software development project conducted by a group of cooperative developers, it is desirable to have some mechanism to support long-duration or cooperative transactions [GRA80, BER87, KOR90, KAI90] so that different users can work on different components and possibly different versions of the software system concurrently. The problem of schema evolution (and therefore data migration) and type evolution [ZDO89] should also be considered if we allow objects of kernel classes (e.g., class, associations, methods, and rules) to be versionable.

9.2.2 Prototyping Environment

Extended GTool. In order to support the whole life-cycle of software development, current implementation of GTool [LAM92] must be extended to include the following: (1) graphic-to-K translator, which will translate the graphic input of the structural and behavioral schemas into corresponding K codes for execution, (2) a syntax-directed editor that will facilitate the writing of K methods and rules, (3) an extended browser that could display different versions of an object (e.g., a class object representing both the specification and implementation of a class), (4) an extended graphic OQL tool that allows the user to specify ad-hoc queries against both user-defined schemas and system-defined kernel schema, (5) an extensible association display facility that could display any user-defined new association types in the schema automatically, and (5) an expert system shell that could facilitate the selection of existing software components to improve reusability.

Prototype execution tool. In order to test the functionalities of a prototype system, certain prototype execution tool is needed to allow the users to interactively select any part of a complex system and execute the selected sub-system. During the execution of the sub-system, incomplete information should be automatically handled by the prototyping tool by either looking up a pre-defined input-output table or

asking the user to enter some test data so that the execution can be continued.

Debugger and performance monitor. A sophisticated debugger is needed to trace the execution of a K program and display the tracing on the graphic user interface. It should also be able to gather some statistic information (e.g., CPU time) about the program execution so that the user can test not only the functionalities but also the performance of the prototype system. It would also be a useful feature if the debugger can "interpret" a K program via a built-in interpreter without using the K compiler. All the above tools will be well-integrated to serve as a KBMS-supported evolutionary prototyping environment.

APPENDIX A SYNTAX SUMMARY OF K

This syntax summary presents the complete syntax of K. The actual lex/yacc specification can be found in Arroyo [ARR92].

Note that the following symbols are meta-symbols belonging to the BNF formalism, and not symbols of K unless explicitly quoted.

`::=` `{` `}` `[` `]`

The square brackets denote optional occurrence of the enclosed symbols, and the curly brackets denote possible repetition of the enclosed symbols zero or more times. All the reserved words are underlined.

Backus-Naur Form (BNF)

```
identifier ::= letter {symbol}
letter ::= upper_case_letter | lower_case_letter
symbol ::= letter | digit | # | _
number ::= integer | real
integer ::= digit {digit}
real ::= integer.integer
string ::= "{character}"
```

```

program ::= definition {definition}

definition ::= class_definition |
               program_definition |
               include_statement

class_definition ::=
    class_type identifier is
        [assoc_section]
        [method_section]
        [rule_section]
        [implementation_section]
    end [identifier];

class_type ::= entity_class | domain_class

assoc_section ::=
    associations : assoc_definition {assoc_definition}

assoc_definition ::=
    aggregation_definition | other_assoc_definition

aggregation_definition ::=
    aggregation of
        encapsulation : aggregation_specs
        {encapsulation : aggregation_specs}

encapsulation ::= public | protected | private

aggregation_specs ::= aggregation_spec {aggregation_spec}

aggregation_spec ::= identifier : class_spec;

class_spec ::= [constructor of] identifier

constructor ::= set | list | array '[' integer ']'

other_assoc_definition ::=
    assoc_type of class_list;

assoc_type ::=
    generalization | specialization | friend | using |
    sequential | parallel | synchronization | testing |
    context_looping

class_list ::= identifiers

identifiers ::= identifier {, identifier}

method_section ::=
    methods : method_definitions

```



```

method_definitions ::=
    encapsulation : method_specs
    {encapsulation : method_specs}

method_specs ::= method_spec {method_spec}

method_spec ::= method_head | operator_head;

method_head ::=
    method identifier ( [var_specs] ) : class_spec

operator_head ::=
    operator overloadable_op ( [var_specs] ) : class_spec

var_specs ::= var_spec {, var_spec}

var_spec ::= identifier : class_spec

overloadable_op ::=
    logical_operator |
    relational_operator |
    arithmetic_operator |
    set_operator |
    unary_operator |

logical_operator ::= and | or

relational_operator ::=
    = | != | > | >= | < | <=

arithmetic_operator ::= + | - | * | / | mod

unary_operator ::= + | - | not

set_operator ::= + | - | &

rule_section ::= rules : rule_definitions

rule_definitions ::= rule_spec {rule_spec}

rule_spec ::=
    rule identifier is
    triggered trigger_conds
    [condition guard_condition]
    [action statements]
    [otherwise statements]
    end [identifier];

trigger_conds ::= trigger_cond {trigger_cond}

trigger_cond ::= timing events;

```

```

timing ::= before | after | immediate_after

events ::= event {, event}

event ::= pnew | new | insert | delete | destroy |
         update [class_tag] identifier |
         method_event

method_event ::=
    [class_tag] identifier()

class_tag ::= identifier ::

guard_condition ::= [guards |] target

guards ::= exp {, exp}

target ::= exp

statements ::= statement {statement}

statement ::=
    exp_statement |
    block_statement |
    conditional_statement |
    repetitive_statement |
    flow_statement |
    object_statement |
    rule_statement |
    abort_statement

exp_statement ::= exp;

exp ::=
    single_item |
    dot_exp |
    assignment_exp |
    binary_exp |
    unary_exp |
    array_exp |
    object_exp |
    quantifier_exp |
    parenthesis_exp

single_item ::=
    identifier | method_invocation |
    number | 'character' | string | boolean

method_invocation ::= identifier ([exp_list])

exp_list ::= exp {, exp}

```

```

boolean ::= true | false

dot_exp ::= exp.identifier | exp.method_invocation

assignment_exp ::=
    exp.identifier := exp |
    identifier := exp

binary_exp ::= exp binary_op exp

binary_op ::=
    logical_operator | relational_operator |
    arithmetic_operator | set_operator |
    cast_operator

cast_operator ::= $

unary_exp ::= unary_operator exp

array_exp ::= exp[ exp ]

object_exp ::=
    new instance_spec |
    pnew instance_spec |
    exp insert instance_spec

instance_spec ::= identifier( [assignments] )

assignments ::=
    assignment_exp {, assignment_exp}

quantifier_exp ::=
    existential_exp |
    universal_exp

existential_exp ::=
    exist var_list in pattern [suchthat exp]

universal_exp ::=
    forall var_list in pattern suchthat exp

var_list ::= identifier {, identifier}

parenthesis_exp ::= ( exp )

pattern ::=
    class_pattern |
    linear_pattern |
    parenthesis_pattern |
    branching_pattern

```

```

class_pattern ::=
    [range_var :] identifier [filter]

range_var ::= identifier

filter ::= '[' exp ']'

linear_pattern ::= pattern link pattern

link ::= assoc_op direction link_id

assoc_op ::= associate | non_associate

associate ::= *

non_associate ::= !

direction ::= > | <

link_id ::= A_link | G_link

A_link ::= '[' identifier ']'

G_link ::= '[' G ']'

branching_pattern ::=
    pattern branch_op (link_pattern_list)

branch_op ::= and | or

link_pattern_list ::=
    link_pattern {, link_pattern}

link_pattern ::= link pattern

parenthesis_pattern ::= ( pattern )

block_statement ::=
    [local_decl]
    begin statements end;

local_decl ::= local var_specs

conditional_statement ::=
    if_statement | case_statement

if_statement ::=
    if exp
    then statements
    [else statements]
    end_if;

```

```

case_statement ::=
    case when exp do statements
        { when exp do statements }
        [ otherwise do statements ]
    end case;

repetitive_statement ::=
    for_statement |
    while_statement |
    context_looping_statement

for_statement ::=
    for identifier
        from exp until exp
        [ by exp ]
        do statements
    end for;

while_statement ::=
    while exp
        do statements
    end while;

context_looping_statement ::=
    context pattern [ where exp ] [ select identifiers ]
        do statements
    end context;

flow_statement ::=
    return_statement |
    continue_statement |
    break_statement

return_statement ::= return [exp];

continue_statement ::= continue;

break_statement ::= break;

object_statement ::=
    delete_statement | destroy_statement

delete_statement ::= delete exp;

destroy_statement ::= destroy exp;

rule_statement ::=
    activate_statement | deactivate_statement

activate_statement ::= activate identifier;

deactivate_statement ::= deactivate identifier;

```

```
abort_statement ::= abort;  
  
implementation_section ::=  
    implementation : method_implementations  
  
method_implementations ::=  
    method_implementation {method_implementation}  
  
method_implementation ::=  
    method_head is statements end [identifier]; |  
    operator_head is statements end;  
  
program_definition ::=  
    program identifier  
        is statements  
        end [identifier];  
  
include_statement ::=  
    include identifier;
```

APPENDIX B PARTS KNOWLEDGE-BASE EXAMPLE

Atkinson and Buneman [ATK87] proposed a set of four tasks to evaluate the expressiveness of data-base programming languages using a manufacturing company's parts data base. In this Appendix, we illustrate how K can be used to specify and implement this application naturally.

Task 1. Describe the knowledge base. We use entity classes "Part," "CompositePart," and "BasePart" to model the parts, and an entity class "Use" to model the many-to-many relationships among parts. Each part has a part-id and can be either a base part or a composite part. A base part has two attributes "cost" and "mass," and a composite part has an attribute "components," which is a set of "Use" instances. Each Use instance has three attributes "parent," "child," and "quantity" to record the relationship that the parent part uses a certain number of the child part.

```
entity_class Part is
  associations:
    aggregation of
      public: part_id : string;
end Part;
```

```

entity_class BasePart is
  associations:
    specialization of Part;
    aggregation of
      public:
        mass : integer;
        cost : integer;
end BasePart;

```

```

entity_class CompositePart is
  associations:
    specialization of Part;
    aggregation of
      public:
        components : set of Use;
end CompositePart;

```

```

entity_class Use is
  associations:
    aggregation of
      public:
        parent    : Part;
        child     : Part;
        quantity  : integer;
end Use;

```

Task 2. Print the name, cost, and mass of all base parts that costs more than 100. This task can be expressed straight-forwardly by using a context looping statement. Note that we define a program named "task2" to perform this task.

```

program task2 is
  context p:BasePart[cost > 100]
  do p.pid.display(); p.cost.display(); p.mass.display();
  end_context;
end task2;

```

Task 3. Compute the total mass of a part named "master." Note that since K.1 does not directly support late binding, we add the method "TotalMass" to classes "Part," "BasePart,"

and "CompositePart," and use the "cast" operator to implement late binding. The "TotalMass" method of class "Part" behaves like a virtual method in C++, in which we use the "cast" operator to explicitly test whether a given Part instance has a corresponding "BasePart" instance or "CompositePart" instance so that the system can in turn apply the correct "TotalMass" method to the correct instance.

```
method TotalMass() : integer is
  case when BasePart$this != null
    do return BasePart$this.TotalMass();
  when CompositePart$this != null
    do return CompositePart$this.TotalMass();
  end_case;
end TotalMass;
```

The implementation of the "TotalMass" method of class "BasePart" is straight-forward and the "mass" attribute value of the given BasePart is returned.

```
method TotalMass() : integer is
  return this.mass;
end TotalMass;
```

The implementation of the "TotalMass" method of class "CompositePart" is more complex as the total mass of a CompositePart is the sum of the multiplication of the total mass of each of its direct subparts (recursion) and the quantity of that subpart. We use a context looping statement to iterate over each "Use" relationship of the given CompositePart and for each iteration, we apply the "TotalMass" method recursively to compute the total mass of a direct

subpart of the given CompositePart. After all the three "TotalMass" methods have been defined, we can define a program "task3" to perform this task. The computation of total cost can be defined in the same way.

```
method TotalMass() : integer is
  local total : integer;
  begin
    total := 0;
    context this *>[components] u:Use *>[child] p:Part
      /* get immediate subparts */
      do total := total + (p.TotalMass()) * (u.quantity);
    end_context;
    return total;
  end;
end TotalMass;
```

```
program task3 is
  context p:Part[part_id = "master"]
    do p.TotalMass().display();
  end_context;
end task3;
```

Task 4. Record a new manufacturing step in the knowledge base, i.e., how a new composite part is manufactured. This task can be simply performed by using the "new" or "pnew" operator to create a new transient or persistent CompositePart instance, respectively. As an example, suppose a new part "P100" uses 2 "P123" part and 5 "N386" part. We define a program task4 to perform this task.

```
program task4 is
  local cp : CompositePart; uses : set of Use;
  begin
    uses := null; /* initialize */
    cp := pnew CompositePart(pid := "P100");
    /* create the new part */
```

```

context p:Part[pid = "P123"]
  do uses := uses + pnew Use(parent := cp,
                             child := p,
                             quantity := 2);
end_context;

context p:Part[pid = "N386"]
  do uses := uses + pnew Use(parent := cp,
                             child := p,
                             quantity := 5);
end_context;

cp.components := uses;
return cp;
end;
end task4;

```

In addition to the above four classical tasks, we further illustrate the expressiveness of K by defining two knowledge base constraints that (i) when a part is deleted, all the parts that uses this part as well as all the related "use" relationships will also be deleted, and (ii) a CompositePart cannot directly or indirectly use itself as a component. The first constraint can be specified by the following "recursive_delete" rule of Part.

```

rule recursive_delete is
  triggered before delete
  condition CompositePart$this != null
    /* this part is a CompositePart */
  action
    context u:Use[child = this]
      /* all the "Use" relationships where this part is
         used as a subpart */
      do delete u.parent;
      delete u;
    end_context;
end recursive_delete;

```

In order to implement the second constraint, we identify a new method "ifUse" of class "CompositePart," which tests if a CompositePart directly or indirectly uses another part as its component.

```
method ifUse( otherPart : Part): boolean is
  local uses : set of uses;
  begin
    uses := this.components;
    context u:uses
      do if u.child = otherPart /* directly uses otherPart */
        then return true;
        else return u.child.ifUse(pid) /* recursion */
      end_if;
    end_context;
    return false;
  end ifUse;
```

After the "ifUse" method is defined, we can specify the constraint as the "recursive_part" rule of class "Use." Note that in the rule condition, we use a guard condition to first make sure if the parent part of a "Use" relationship is a CompositePart, then we apply the "ifUse" method to test if the parent part uses itself. Once defined and compiled, the knowledge rules will be automatically triggered by the system to keep the knowledge base in a consistent state.

```
rule recursive_part is
  triggered after insert
  condition CompositePart$(this.parent) != null |
    CompositePart$(this.parent).ifUse(this.parent)
  action delete this; /* delete this relationship */
end recursive_part;
```

REFERENCES

- [ACM91] ACM, Communication of ACM, Special Issue on Next-Generation Database Systems, Vol. 34(10), October 1991.
- [AGR89] Agrawal, R. and Gehani, N., "ODE (Object Database and Environment): The Language and the Data Model," Proc. 1989 ACM SIGMOD Int'l Conf. on Management of Data, 1989, pp. 36-45.
- [AHO86] Aho, A.V., Sethi, R., and Ullman, J.D., Compilers-Principles, Techniques, and Tools, Addison-Wesley Pub. Co., Reading, MA, 1986.
- [ALA89] Alashqur, A., Su, S. and Lam, H., "OQL--A Query Language for Manipulating Object-oriented Databases," Proc. of 15th Int'l Conf. Very Large Data Bases, Amsterdam, Netherlands, August, 1989, pp. 433-442.
- [ALA90] Alasqur, A.M., Su, S.Y.W., and Lam, H., "A Rule-based Language for Deductive Object-Oriented Databases," Proc. of the sixth Int'l Conference on Data Engineering, Los Angeles, Feb. 5-9, 1990.
- [ALB85] Albano, A., Cardelli, L. and Orsini, R., "Galileo: A Strongly-typed, Interactive Conceptual Language," ACM Transactions on Database Systems, 10(2), June 1985, pp. 230-260.
- [AND87] Andrews, T. and Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment," Proc. 2nd Int'l Conf. on OOPSLA, October 1987, pp. 430-440.
- [ANN91] Annevelink, J., "Database Programming Languages: A Functional Approach," ACM SIGMOD Int'l Conf. on Management of Data, 1991, pp. 318-327.
- [ARR92] Arroyo, J., "The Design and Implementation of K: A Next-Generation Knowledge-Base Programming Language," Master's Thesis, Electrical Engineering Department, University of Florida, Gainesville, August 1992.

- [ATK83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, P.W., and Morrison, R., "An Approach to Persistent Programming," Computer Journal, Vol. 26(4), 1983.
- [ATK87] Atkinson, M.P. and Buneman, P.O., "Types and Persistence in Database Programming Languages," ACM Computing Surveys, July 1987, pp. 105-190.
- [ATK90] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S., "The Object-Oriented Database System Manifesto," in Deductive and Object-Oriented Databases, Kim, W., Nicolas, J.M., and Nishio, S., (eds.), Elsevier Science Pub. Co., New York, 1990, pp. 223-240.
- [BAL81] Balzer, R., Final Report on GIST, Information Science Institute, Univ. of Southern California, Marina Del Rey.
- [BAL85] Balzer, R., "A 15 Year Perspective on Automatic Programming," IEEE Trans. on Soft. Engr., Vol. SE-11(11), November 1985, pp. 1257-1268.
- [BAL82] Balzer, R., Goldman, N., and Wile, D., "Operational Specifications as the Basis for Rapid Prototyping," ACM SIGSOFT Soft. Eng. Notes, Vol. 7(5), December 1982, pp. 3-16.
- [BEE88] Beech, D. and Mahbod, B., "Generalized Version Control in an Object-Oriented Database," Proc. 4th Int'l Conf. on Data Engineering, February 1988, pp. 14-22.
- [BER87] Bernstein, P., "Database System Support for Software Engineering," Proceedings of the International Conference on Software Engineering, 1987, pp. 161-178.
- [BER88] Berzins, L., Luqi, Q., and Yeh, R., "PSDL: A Prototyping Language for Real-Time Software," IEEE Trans. on Software Engineering, Vol. 14(10), October 1988, pp. 1409-1423.
- [BLA90] Blakeley, J.A., Thompson, C.W., and Alascur, A.M., "OQL[X]: Extending a Programming Language X with a Query Capability," Technical Report 90-07-01, Information Technologies Laboratory, Texas Instruments Inc., Dallas, Texas, 1990.

- [BLO87] Bloom, T. and Zdonik, S.B., "Issues in the Design of Object-oriented Database Programming Languages," Proc. 2nd Int'l Conf. on OOPSLA, October 1987, pp. 441-451.
- [BOE88] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," Computer, May 1988, pp. 61-72.
- [BOO90] Booch, G., Object-Oriented Design with Applications, Menlo Park, CA: Benjamin-Cummings, 1990.
- [BRO83] Brodie, M. and Ridjanovic, D., "On the Design and Specification of Database Transactions," in On Conceptual Modeling, Brodie, M., Mylopoulos, J., and Schmidt, J.W., (eds.), New York: Springer-Verlag, 1983.
- [BUT91] Butterworth, P., Otis, A., and Stein, J., "The GemStone Object Database Management System," CACM, October 1991, Vol. 34(10), pp. 64-77.
- [CAC90] Cacace, F., Ceri, S., Crespi-Reghizzi, S., Tanca, L., Zicari, R., "Integrating Object-oriented Data Modeling with a Rule-based Programming Paradigm," Proc. ACM SIGMOD 1990, pp. 225-236.
- [CAR90] Carey, M., DeWitt, D., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., and Vandenberg, S.L., "The EXODUS Extensible DBMS Project: An Overview," Readings in Object-Oriented Database Systems, Zdonik, S., and Maier, D., (eds.), San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990, pp. 474-499.
- [CHA89] Chakravorthy, U.S., "Rule Management and Evaluation: An Active DBMS Perspective," SIGMOD RECORD, Vol. 18(3), September 1989, pp. 20-28.
- [CHE76] Chen, P.P., "The Entity-Relationship Model: Toward a Unified View of Data," ACM Trans. on Database Systems, 1976, pp. 9-36.
- [CHU90] Chung, H.S., "Operational Rule Processing in a Prototype OSAM* Knowledge Base Management System," Master's Thesis, Computer and Information Science Department, University of Florida, Gainesville, 1990.
- [CHU91] Chung, L., Katalagarianos, P., Marakakis, M., Mertikas, M., Mylopoulos, J., Vassiliou, Y., "From Information System Requirements to Designs: A

- Mapping Framework," Information Systems, Vol. 16(4), pp. 429-461, 1991.
- [COP84] Copeland, G. and Maier, D., "Making Smalltalk a Database System," Proc. 1988 ACM SIGMOD Int'l Conference on Management of Data, June 1984, pp. 316-325.
- [DAV58] Davis, M., Computability and Unsolvability, McGraw-Hill, New York, 1958.
- [DAY88] Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ladin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M., Livny, M., and Jauhari, R., "The HiPAC Project: Combining Active Databases and Timing Constraints," SIGMOD RECORD, Vol. 17(1), March 1988, pp. 51-70.
- [DAY90] Dayal, U., Hsu, M., and Ladin, R., "Organizing Long-Running Activities with Triggers and Transactions," Proc. 1990 ACM SIGMOD Int'l Conf. on Management of Data, pp. 204-214.
- [DEM78] DeMarco, T., Structured Analysis and System Specification, New York: Yourdon, 1978.
- [DEU91] Deux, O., "The O2 System," CACM, Vol. 34(10), October 1991, pp. 34-48.
- [DOD83] DOD--U.S. Department of Defense, Reference Manual for the Ada Programming Language, U.S. Department of Defense, Washington, D.C., 1983.
- [ELL89] Ellis, C.A. and Gibbs, S.J., "Active Objects: Realities and Possibilities," in Object-Oriented Concepts, Databases, and Applications, Kim, W. and Lochovsky, F.H., (eds.), ACM Press, New York, 1989, pp. 561-572.
- [FIS87] Fishman, D., Beech, D., Cate, H., Chow, E., Connors, T., Davis, J., Derrett, N., Hoch, C., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M., Ryan, T., and Shan, M., "IRIS: An Object-Oriented Database Management Systems," ACM Transactions on Office Information Systems, Vol. 5, No.1, January 1987.
- [GEH91] Gehani, N.H. and Jagadish, H.V., "Ode as an Active Database: Constraints and Triggers," Proc. of 17th Int'l Conf. Very Large Data Bases, 1991.

- [GRA80] Gray, J., "The Transaction Concept: Virtues and Limitations," Proc. 7th Int'l Conf. on VLDB, 1981, pp. 144-154.
- [GUO91] Guo, M.S., Su, S.Y.W., and Lam, H., "An Association Algebra for Processing Object-Oriented Databases," Proc. 7th IEEE Int'l Conf. on Data Engineering, Kobe, Japan, 1991.
- [HAM80] Hammer, M., and Berkowitz, B., "DIAL: A Programming Language for Data Intensive Applications," Proc. ACM SIGMOD Conf. on Management of Data, 1980.
- [HAM81] Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model," ACM Transactions on Database Systems, Vol. 6(3), September 1981, pp. 351-386.
- [HAN89] Hanson, E., "An Initial Report on the Design of Ariel: A DBMS with an Integrated Production Rule System," SIGMOD RECORD, Vol. 18(3), September 1989, pp. 12-19.
- [HSU88] Hsu, M., Ladin, R., and McCarthy, D.R., "An Execution Model for Active Data-base Management Systems," Proc. 3rd Int'l Conf. on Data and Knowledge Bases, 1988.
- [HUD87] Hudson, S.E. and King, R., "Object-Oriented Database Support for Software Environments," ACM SIGMOD Int'l Conf. on Management of Data, 1987, pp. 491-503.
- [HUL87] Hull, R. and King, R., "Semantic Database Modeling: Survey, Application, and Research Issues," ACM Computing Surveys, Vol. 19(3), September 1987, pp. 201-258.
- [JAR90] Jarke, M., Jeusfeld, M., and Rose, T., "A Software Process Data Model for Knowledge Engineering in Information Systems," in Information Systems, Vol. 15(1), pp. 85-116, 1990.
- [KAI90] Kaiser, G.E., "A Flexible Transaction Model for Software Engineering," Proc. of the Sixth Int'l Conference on Data Engineering, Los Angeles, Feb. 5-9, 1990.
- [KAP91] Kappel, G. and Schrefl, M., "Object/Behavior Diagrams," Proc. 7th IEEE Int'l Conf. on Data Engineering, Kobe, Japan, 1991, pp. 530-539.

- [KH086] Khoshfian, S. and Copeland, G., "Object Identity," Proc. ACM OOPSLA'86, pp. 406-414.
- [KIM88] Kim, W., Ballou, N., Chou, H.T., Garza, J.F., and Woelk, D., "Integrating an Object-Oriented Programming System with a Database System," Proc. 3rd Int'l Conf. on OOPSLA, September 1988, pp. 142-152.
- [KOR90] Korth, H.F. and Speegle, G.D., "Long-Duration Transactions in Software Design Projects," Proc. of the Sixth Int'l Conference on Data Engineering, Los Angeles, Feb. 5-9, 1990.
- [KOW79] Kowalski, R., "Algorithm = Logic + Control," Comm. of ACM, July 1979, pp. 424-475.
- [KUN89] Kung, C.H., "Conceptual Modeling in the Context of Software Development," IEEE Transactions on Software Engineering, Vol. 15(10), October 1989, pp. 1175-1187.
- [LAM89a] Lam, H., Su, S. and Alashqur, A., "Integrating the Concepts and Techniques of Semantic Modeling and the Object-Oriented Paradigm," Proc. 13th Int'l Computer Software & Applications Conference (COMPSAC), October 1989, pp. 209-217.
- [LAM89b] Lam, H., Xia, D., Qiu, J., Wu, P., Chen, H., Pant, S., Geum, S., and Su, S., "Prototype Implementation of an Object-Oriented Knowledge Base Management System," extended abstract, Proc. the Second Florida Conference on Productivity through Computer Integrated Engineering and Manufacturing, November, 1989, pp. 68-70.
- [LAM92] Lam, H., Su, S.Y.W., Ruhela, V., Pant, S., Ju, S.M., Sharma, M., and Prasad, N., "GTOOLS: An Active Graphical User Interface Toolset for an Object-Oriented KBMS," to appear in the International Journal of Computer Science and Engineering, 1992.
- [LAM91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D., "The ObjectStore Database System," CACM, Vol. 34(10), October 1991, pp. 50-63.
- [LAW90] Law, S.F., "Object-Oriented Design and Implementation of a Kernel Object Manager," Master's Thesis, Electrical Engineering Department, University of Florida, Gainesville, 1990.

- [LEC89] Lecluse, C. and Richard, P., "The O2 Database Programming Language," Proc. 15th VLDB Conf., Amsterdam, August 1989, pp. 411-422.
- [LIE86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," Proc. 1st Int'l Conf. on OOPSLA, September 1986, pp. 214-223.
- [LIN88] Lingat, J. and Rolland, C., "Rapid Application Prototyping: The Proquel Language," Proc. 14th VLDB Conference, Los Angeles, 1988, pp. 206-217.
- [LIS77] Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., "Abstraction Mechanisms in CLU," CACM, 20(8), August 1977, pp. 564-576.
- [LOH91] Lohman, G.M., Lindsay, B., Pirahesh, H., Schiefer, K.B., "Extensions to Starburst: Objects, Types, Functions, and Rules," CACM, Vol. 34(10), October 1991, pp. 94-109.
- [MAI89] Maier, D., "Why Database Languages Are a Bad Idea," in Workshop on Database Programming Languages, Bancilhon, F. and Buneman, P. (eds.), Addison-Wesley Pub. Co., Reading, MA, 1989.
- [MAI86] Maier, D., Stein, J., Otis, A. and Purdy, A., "Development of an Object-Oriented DBMS," Proc. OOPSLA'86, September 1986.
- [MAR90] Markowitz, V.M., "Representing Processes in the Extended Entity-Relationship Model," Proceedings of the 6th International Conference on Data Engineering, 1990, pp. 103-110.
- [MEY88] Meyer, B., Object-Oriented Software Construction, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [MIL88] Mills, H.D., "Stepwise Refinement and Verification in Box-Structured Systems," IEEE Computer, June 1988, pp. 23-36.
- [MOS81] Moss, J., "Nested Transactions: An Approach to Reliable Distributed Computing," MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.
- [MYL80] Mylopoulos, J., Bernstein, P.A. and Wong, H.K.T., "A Language Facility for Designing Database-Intensive Applications," ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980, pp. 185-207.

- [NAS73] Nassi, I. and Shneiderman, B., "Flowchart Techniques for Structured Programming," SIGPLAN Notices 8(8), 1973.
- [ONT91] Ontologic Inc., "ONTOS 2.1 Product Description," Burlington, MA, 1991.
- [PAR83] Partsch, H. and Steinbruggen, R., "Program Transformation Systems," ACM Computing Surveys, Vol. 5(3), September 1983, pp. 199-236.
- [PET81] Peterson, J.L., Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [PHI91] Phipps, G. and Derr, M., "Glue-Nail: A Deductive Database System," in ACM SIGMOD Int'l Conf. on Management of Data, 1991, pp. 308-317.
- [REI87] Reiss, S., "Working in the Garden Environment for Conceptual Programming," IEEE Software, November 1987, pp. 16-27.
- [RIC87] Richardson, J. and Carey, M., "Programming Constructs for Database System Implementation in EXODUS," Proc. 1987 ACM SIGMOD Int'l Conf. on Management of Data, 1987, pp. 208-219.
- [RIC91] Richardson, J. and Schwartz, P., "Aspects: Extending Objects to Support Multiple, Independent Roles," ACM SIGMOD Int'l Conf. on Management of Data, 1991, pp. 298-307.
- [ROW79] Rowe, L.A. and Shoens, K.A., "Database Abstractions, Views, and Updates in RIGEL," Proc. ACM SIGMOD Conf. on Management of Data, Boston, May 1979, pp. 71-81.
- [ROW87] Rowe, L.A. and Stonebraker, M., "The POSTGRES Data Model," Proc. 13th Int'l Conf. Very Large Data Bases, Brighton, England, September 1987, pp. 83-96.
- [RUM87] Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language," Proc. OOPSLA '87, Orlando, FL, 1987. pp. 466-481.
- [SCH86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C., "An Introduction to Trellis/Owl," Proc. 3rd Int'l Conf. on OOPSLA, September 1988, pp. 9-16.

- [SCH77] Schmidt, J.W., "Some High Level Language Constructs for Data Type Relation," ACM Trans. on Database Systems, September 1977, Vol. 2(3), pp. 247-281.
- [SHI81] Shipman, D.W., The Functional Data Model and the Data Language DAPLEX," ACM Transactions on Database Systems, Vol. 6(3), September 1981.
- [SHY91] Shyy, Y.M. and S.Y.W. Su, "K: a High-Level Knowledge Base Programming Language for Advanced Database Applications," Proc. 1991 ACM SIGMOD Int'l Conference on Management of Data, Denver, Colorado, May 29-31, 1991, pp. 338-347.
- [SIG90] SIGMOD, Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto," SIGMOD Record, Vol. 19(3), September 1990, pp. 31-44.
- [SIL91] Silberschatz, A., Stonebraker, M., and Ullman, J., "Database Systems: Achievements and Opportunities," CACM, Vol. 34(10), October 1991, pp. 110-120.
- [SIN90] Singh, M., "Transaction Oriented Rule Processing in an Object-Oriented Knowledge Base Management System," Master's Thesis, Department of Electrical Engineering, University of Florida, Gainesville, 1990.
- [SMI85] Smith, D.R., Kotik, G.B., and Westfold, S.J., "Research on Knowledge-Based Software Environments at Kestrel Institute," IEEE Trans. on Software Engineering, Vol. 11(11), November 1985, pp. 1278-1295.
- [SMI77] Smith, J. and Smith, C., "Database Abstraction: Aggregation and Generalization," ACM Trans. on Database Systems, Vol. 2(2), 1977.
- [SMI83] Smith, J.M., Fox, S., and Landers, T., ADAPLEX: Rational and Reference Manual, 2nd edition, Computer Corporation of America, Cambridge, MA, 1983.
- [STE86] Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations," AI Magazine, 6(4), 1986, pp. 40-64.
- [STE89] Stein, L.A. and Zdonik, S.B., "Clovers: The Dynamic Behavior of Types and Instances," Technical Report CS-89-42, Computer Science Department, Brown University, Providence, RI, November 1989.

- [STO91] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next Generation Database Management System," CACM, Vol. 34(10), October 1991, pp. 78-92.
- [STR86] Stroustrup, B., The C++ Programming Language, Addison-Wesley Pub. Co., Reading, MA, 1986.
- [SU86] Su, S.Y.W., "SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases," Journal of Information Sciences, 29, 1983, pp. 151-199.
- [SU89a] Su, S.Y.W., "Extensions to the Object-Oriented Paradigm," Proc. 13th Int'l Computer Software & Applications Conference (COMPSAC), October 1989, pp. 197-199.
- [SU91] Su, S.Y.W., and Alashqur, A.M., "A Pattern-based Constraint Specification Language for Object-Oriented Databases," Proc. of IEEE COMPCON'91, San Francisco, Feb. 25-March 1, 1991.
- [SU89b] Su, S.Y.W., Krishnamurthy, V. and Lam, H., "An Object-Oriented Semantic Association Model OSAM*," in Artificial Intelligence Manufacturing Theory and Practice, Kumara, S.T., Soyster, A.L., and Kashyap, R.L., (eds.), Industrial Engineering and Management Press, American Inst. of Indus. Engr., Norcross, GA, 1989, Chap. 17, pp. 463-494.
- [SU92] Su, S.Y.W. and Shyy, Y.M., "An Object-Oriented Knowledge Model for KBMS-Supported Evolutionary Prototyping of Software Systems," Technical Report TR92-011, Computer and Information Science Department, University of Florida, Gainesville, May 1992.
- [TSU86] Tsur, S. and Zaniolo, C., "LDL: a Logic Based Data Language", Proc. of Int'l Conf. on VLDB, Kyoto, August, 1986.
- [WAS81] Wasserman, A.I., Sherertz, D.D., Kersten, M.L., van de Riet, R.P., Dippe, M.D., "Revised Report on the Programming Language PLAIN," ACM SIGPLAN Notices, Vol. 16(5), May 1981, pp. 59-80.
- [WIL90a] Wile, D., "Adding Relational Abstraction to Programming Languages," Proc. ACM SIGSOFT Int'l

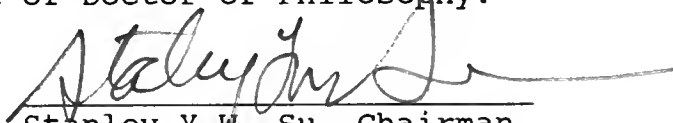
Workshop on Formal Methods in Software Development,
Napa, CA, May 9-11, 1990, pp. 128-139.

- [WIL90b] Wilkinson, K., Lyngbaek, P., and Hassan, W., "The Iris Architecture and Implementation," IEEE Transactions on Knowledge and Data Engineering, Vol. 2(1), March 1990, pp. 63-75.
- [WIR76] Wirth, N., Programs = Algorithms + Data Structures, Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [WIR90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [YAS91] Yassen, R., Su, S.Y.W., and Lam, H., "An Extensible Kernel Object Management System," Proc. ACM SIGPLAN OOPSLA'91, 1991, pp. 247-263.
- [ZDO89] Zdonik, S., "Object-oriented Type Evolution," in Advances in Database Programming Languages, Bancilhon, F. and Buneman, P. (eds.), Addison-Wesley Pub. Co., Reading, MA, 1989.
- [ZDO86] Zdonik, S. and Wegner, P., "Language and Methodology for Object-Oriented Database Environments," Proc. 19th Annual Hawaiian Conference on Systems Science, 1986.

BIOGRAPHICAL SKETCH

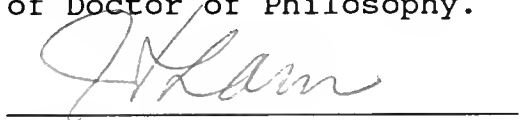
Yuh-Ming Shyy was born in Taipei, Taiwan, in 1959. He received his B.S. degree in electrical engineering from National Taiwan University in 1982, and the Sc.M. degree in computer science from Brown University, Providence, Rhode Island, in 1987. He is currently working toward the Ph.D. degree in the Computer and Information Science Department of the University of Florida in Gainesville. He has been a research assistant in the Database Systems Research and Development Center of the department since 1988. His research interests are in object-oriented next-generation data-base systems, object-oriented data-base programming languages, object-oriented software design and management, data-base support for software engineering, rapid prototyping, active data-base systems, office information systems, and distributed heterogeneous data-base systems for future information networks.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



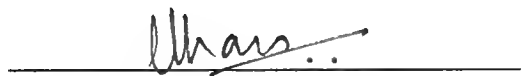
Stanley Y.W. Su, Chairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



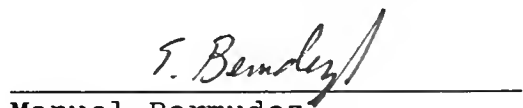
Herman Lam
Associate Professor of
Computer and Information
Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Sharma Chakravarthy
Associate Professor of
Computer and Information
Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Manuel Bermudez
Associate Professor of
Computer and Information
Sciences

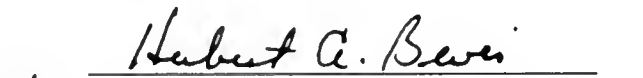
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Thomas Bullock
Professor of
Electrical Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August, 1992



for Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School